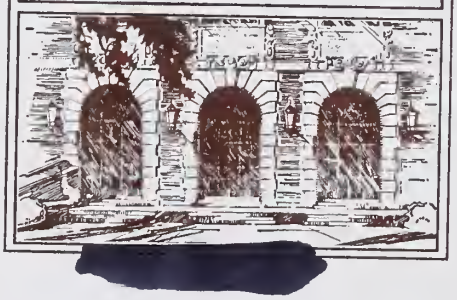
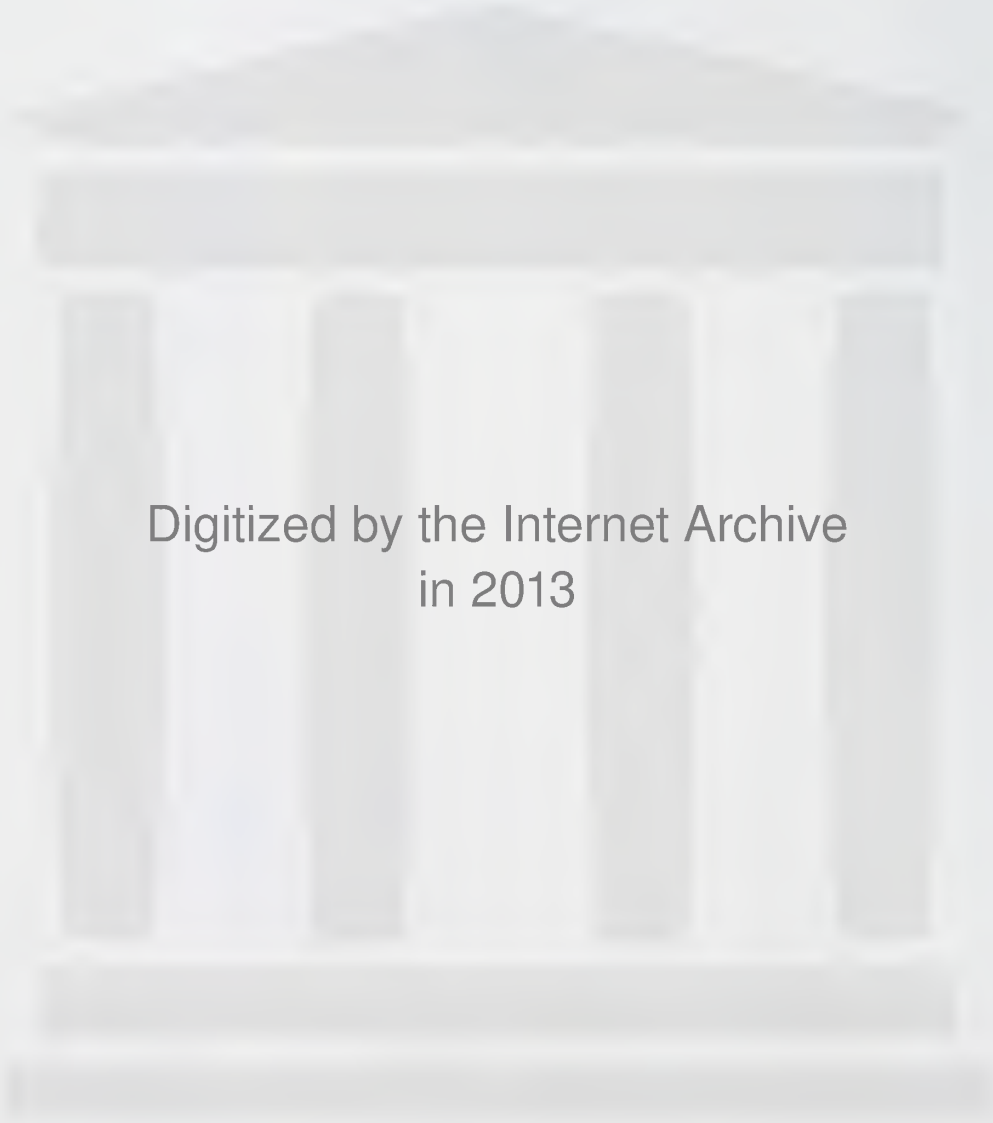




Report No. UIUCDCS-R-75-747 NSF - OCA - DCR73-07980 A02 - 0000011

ANALYSIS AND DESIGN OF INTERLEAVED MEMORY SYSTEMS

by

DONALD YI-CHUNG CHANG

August 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-75-747

ANALYSIS AND DESIGN OF INTERLEAVED MEMORY SYSTEMS*

by

DONALD YI-CHUNG CHANG

August 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF DCR73-07980 A02 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, August 1975.

ACKNOWLEDGMENT

I would like to express my sincere thanks to my advisor, Professor D. J. Kuck, for his constant encouragement, patience, and guidance. I would also like to thank Professor D. H. Lawrie for his valuable discussions and suggestions. Special thanks go to Mrs. Vivian Alsip for her help in preparing this thesis.

I also wish to thank my wife, Li, for her excellent typing job as well as her long time assistance during my study at the University of Illinois.

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1 What is an Interleaved Memory System

A traditional computer system consists of five components - a processing unit, a primary memory, a control unit, an input/output subsystem, and a secondary memory. The memory is the central element of the whole system, in the sense that it is the source or destination of all information flowing to or from the other portions of the system. So the memory hierarchy and memory management are two extremely important subjects in computer design since they greatly influence the throughput and utilization of a computer system. For years, memory design has been a core problem of computer architecture.

Some computer systems have only one main memory module. In every memory cycle, only one word in the memory can be accessed. Since the memory speed is usually slower than the processor speed, in most cases they are incompatible. The processor is often lying idle waiting for the memory. Hence the throughput and utilization of the system are reduced significantly by the slowness of the memory.

Naturally, people would try to solve this problem using hardware, that is, to construct faster memory by using faster circuits. A lot of effort has been spent in this area trying to invent a faster logic component that can be used as a memory unit. Right now, modern memory technology can produce a memory with less than a hundred nanoseconds cycle time, for example a bipolar memory. However, faster memory is expensive and cost becomes a discouraging factor, especially in a large computer system.

A simple but better way is just to break the whole memory into several pieces and attach a decoder to each submodule. Then we would be able to access several words at the same time if we have a fancier control unit. Thus we effectively reduce the memory cycle time by several times. This scheme is called an interleaved memory system. The word "interleaved" comes from the fact that we can access several memory modules at the same time in a random fashion.

The interleaved memory systems have many beneficial effects: (1) each module is smaller, hence the memory cycle time will be slightly reduced due to the shorter address decoding time, (2) several memory can be accessed at the same time, (3) they can be implemented by using cheaper and easy to be replaced elements, hence more suitable for the modern trend of modular computer systems.

In a modern high-speed system, such as ILLIAC IV or B6700, the main memory is divided into several modules. This modularized memory system also has the advantages of easy to be replaced and easy to expand. Some systems even use the bus structure to overlap the memory operations, thus reducing the memory cycle time effectively. So modularity is very similar to interleaving in some sense, since they achieve some common goals.

We believe that the interleaved and modular systems will become the main trend of memory design in the future. That is the reason we study this problem here.

1.2 Thesis Organization

In Chapter 2 of this thesis, we will describe some of the work other people have done in this area and describe their models of interleaved memory systems. Most of this work has been done analytically, we will show how they got their equations and also show our further results.

One thing no one else has explicitly considered before is the data dependency between requests. It is very important in real machines. In Chapter 3, we will discuss several possible kinds of data dependency in the real programs. Data dependency between requests will constrain the usefulness of a model, and so it should be considered carefully.

In Chapter 4, we will present four models we have worked on. A description of each model will be given first, then we will show the performances we have measured. All the models, including some of those studied by other people, will be compared on the same basis in Chapter 5.

Chapter 6 contains some circuit designs and system layouts of our models. Although our designs are very simple, they can show the feasibility of our models.

The last chapter is the conclusion of our work.

In order to give a concise presentation, we will put all the messy details, such as the derivation of equations, in the appendix which follows the conclusion.

2. HISTORY

2.1 Early Works

The idea of using multiple memory modules to increase system throughput was thought of by people more than ten years ago. In 1964, Ivan Flores wrote a paper [1] to analyze a multiple-bank memory system. He used queueing theory arguments to derive the waiting-time factor of the system and show how waiting time varies with respect to several other parameters. As you will see, the waiting-time factor is similar to the bandwidth which we will define very soon.

Later, Michael Flynn [2] used a different approach to tackle the same problem. He studied a lot of recorded address traces of real programs and measured the "true" waiting time and the bandwidth of those programs. He defined bandwidth as the retrieval rate of words from memory, that is, the average number of words that can be accessed in one memory cycle. This is what most people are interested in.

No doubt, Flynn's results are more realistic than Flores'. However, they only reflect the performance of the programs he analyzed. If we want to know the general performance of a certain system we need to study a rather large number of programs of several different characteristics. This is very time-consuming. Work in this area is usually done by defining a model and then analyzing it analytically or by simulation. Although most of the models people design are very abstract, they still give some flavor of how a factor will influence the system performance.

2.2 Hellerman's Model

In [3], H. Hellerman presented a very simple interleaved memory system as shown in Figure 1, and gave the system performance in terms of the average relative bandwidth. The definition of the average relative bandwidth is the same as what Flynn used in his system. Later, a lot of people began to study interleaved memory systems in Hellerman's way and use the same definition of bandwidth as the system measurement. We will start with Hellerman's model and give more detailed description of the works other people have done along this line.

Hellerman's model is an m memory module system with a single string of requests coming in. Each request is an integer from 0 to m-1 that specifies a specific module. All requests are assumed to be equally likely, or generated randomly, and the input stream always contains more than m requests. Then the average relative bandwidth can be identified as the average length of a string of distinct integers.

The probability of a string of distinct integers with length k is:

$$P_k = \frac{k(m-1)!}{m^k(m-k)!}$$

since the probability of choosing first request is 1, second request is $\frac{m-1}{m}$, third request is $\frac{m-2}{m}$,..., kth request is $\frac{m-k+1}{m}$, and (k+1)th request is $\frac{k}{m}$ since it must be equal to one of the previous k requests. Multiply them together and you get the above result. Then, by probability definition, the average length (and hence the average relative bandwidth) is:

$$B_{av.} = \sum_{k=1}^{m} kP_k \qquad (2.1)$$

Substituting P_k into equation (2.1) we get:

$$B_{av.} = \sum_{k=1}^{m} \frac{k^2(m-1)!}{m^k(m-k)!} \qquad (2.2)$$

Equation (2.1) is adopted as the definition of bandwidth by most people, and the efforts focus on finding the proper P_k for each model.

When $1 \leq m \leq 45$, Hellerman found a good numerical approximation to equation (2.2) to be $m^{0.56}$, or approximately $\sqrt{m}$. The error is no more than 4.3%. This is the square root concept that has been generally accepted. That is, when you increase the number of memory modules in your system, the bandwidth will grow as the square root of m. However, in Chapter 4, we will show that this is not quite true.

Very recently, Knuth and Rao have shown in [4] that a nice closed form to equation (2.2) can be mathematically proved to be:

$$B_{av.} = \sqrt{\frac{\pi m}{2}} + \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2m}} + O(m^{-1})$$

which tells that the average relative bandwidth of Hellerman's model is indeed asymptotic to the square root of m.

One thing Hellerman did not say about his model is how many processors he used to generate those requests. Apparently, he implicitly assumed that those requests were generated by one single processor, otherwise there is no reason why he should stop at the first conflict. If this is the case, the bandwidth should not grow as the square root function of m indefinitely. Since no matter how fast the processor is it can only generate a finite number of requests per memory cycle, say M requests and M may be very large. When $m > M$, the bandwidth should not grow any more since the request supply is finite. So a reasonable graphic representation

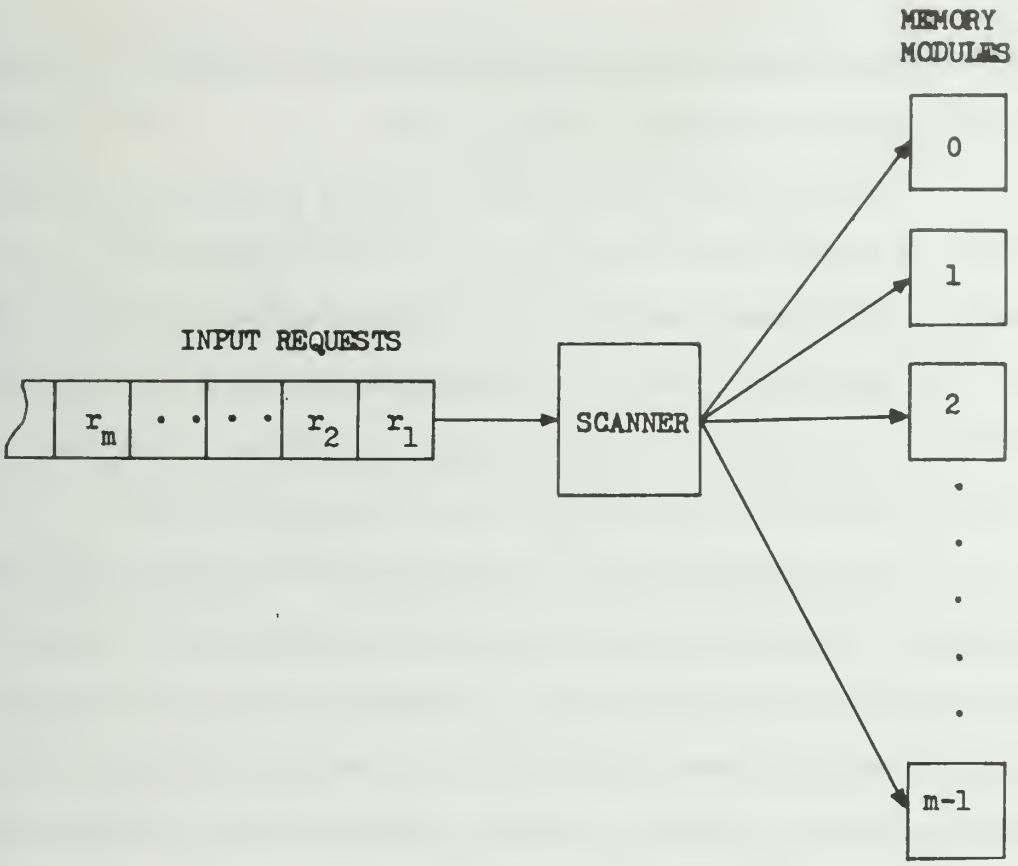


Figure 1. Hellerman's Model

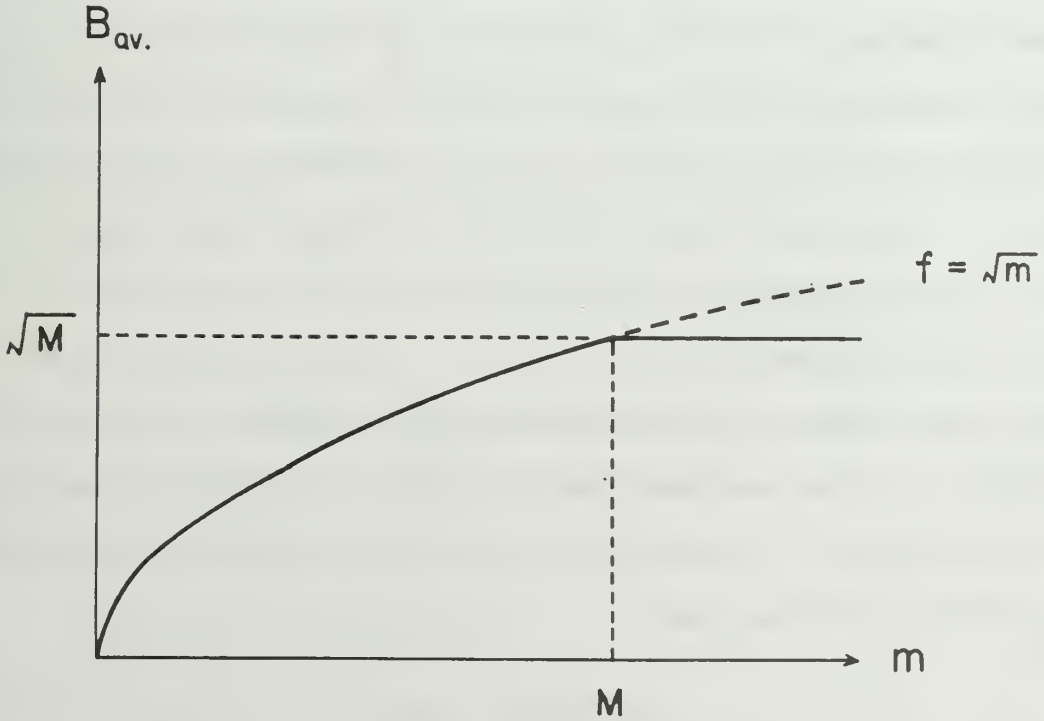


Figure 2. Bandwidth Curve of Hellerman's Model

of Hellerman's model should look like Figure 2. Beyond the point m=M, the curve becomes a horizontal line $B_{av.} = \sqrt{M}$.

On the other hand, you may say that in real machine design the number of memory modules used will be at most a few hundred, so we still should accept square root of m as the fact. However, it is very easy to see that Hellerman's model is only good when there is no logical dependency between the requests. We will discuss this problem in the next chapter. The we will propose a more realistic way of designing a model.

In a 1973 paper [5], G. Burnett and E. Coffman generalized Hellerman's model and showed a more interesting result. They used the same model, the same definition of bandwidth, and almost the same assumption about the requests except they put two probability parameters α and β in the input request stream. Then they show how α would influence the bandwidth in addition to m.

They assumed that the probability of a request addressing the next module in sequence (modulo m) will be α and the probability of addressing any other module out of sequence will be β. Where $\beta = (1-\alpha)/(m-1)$. Or formally, let $r_1, r_2, \ldots, r_i, r_{i+1}, \ldots$ denote the input request stream, then

$$P(r_1 = j) = 1/m, \qquad j \in K_m = \{0,1,\ldots,m-1\}$$

$$P(r_{i+1} = (r_i+1) \bmod m) = \alpha, \qquad i = 1,2,3,\ldots.$$

$$P(r_{i+1} = n) = \beta, \qquad n \in K_m \text{ and } n \neq (r_i+1) \bmod m$$

For example, when m=8 the 8-length sequence 06723145 would have probabilty $\frac{1}{8}\alpha^3\beta^4$. By the pigeonhole principle, the longest possible sequence would be of length m. The successive addressing is called an α-transition and the other a β-transition.

By using another form of equation (2.1), they define their bandwidth as:

$$B_{av.} = \sum_{k=1}^{m} P_k(w \geq k)$$

where $P_k(w \geq k)$ is the probability of a sequence with length at least k. Then they got the bandwidth equation:

$$B_{av.} = \sum_{k=1}^{m} \sum_{j=0}^{k-1} \alpha^j \beta^{k-1-j} C_m(j,k)$$

where $C_m(j,k)$ is the total number of k-length sequences with j α-transitions, k-1-j β-transitions and $r_1=0$. It is very easy to show that the inner summation is the probability $P_k(w \geq k)$. Since the number of sequences begin with 0 is the same as the number of sequences begin with any other number, they only count a special class of sequences and cancel 1/m in their equation. Note that a k-length sequence with j α-transitions should have the probability $1/m\ \alpha^j\ \beta^{k-1-j}$.

The counting of $C_m(j,k)$ is a very interesting combinatorial problem. Unfortunately, they solved this combinatorial problem by using a state transition argument and reached a rather complicated result.

They first derived a theorem which says that for a fixed ordering of j α-transitions, the total number of k-length sequences with j α-transitions and beginning with a 0 is the same as that for any other ordering. Let $C_m^0(j,k)$ be the number of k-length sequences that begin with a 0 and with all α-transitions occurring at the first j transitions. Then this theorem can be written as:

$$C_m(j,k) = \binom{k-1}{j} C_m^0(j,k) \qquad (2.3)$$

where $\binom{k-1}{j}$ is the total number of orderings.

Then by using the following three facts:

$$C_m^0(j,k) = C_{m-j}^0(0,k-j) \tag{2.4}$$

$$(m-1)_{k-1} = \sum_{j=0}^{k-1} C_m(j,k) \tag{2.5}$$

$$C_m(0,k) = C_m^0(0,k) \tag{2.6}$$

they first got the recursive solution for $C_m^0(0,k)$ by equations (2.5), (2.6) and (2.3):

$$C_m(0,k) = (m-1)_{k-1} - \sum_{j=1}^{k-1} \binom{k-1}{j} C_{m-j}^0(0,k-j)$$

Where $(m-1)_{k-1}$ is the falling factorial of k-1 terms, ie. $(m-1)(m-2)\ldots(m-k+1)$. Substituting equation (2.4) into (2.3), they got the solution for $C_m(j,k)$:

$$C_m(j,k) = \binom{k-1}{j} C_m^0(j,k)$$

$$= \binom{k-1}{j} C_{m-j}^0(0,k-j)$$

where $C_{m-j}^0(0,k-j)$ can be evaluated by the above recursive equation. The boundary conditions are:

$$C_m^0(0,1) = 1$$

$$C_m^0(k-1,k) = 1$$

Although they presented an evaluation ordering for these numbers so that no number will be calculated twice, their equation still takes a long time to solve.

Later in a short note [6], H. Stone used an inclusion-exclusion argument to get a direct solution for $C_m^0(j,k)$. If a_i is defined to be the property that the ith transition is an α-transition, then

$$C_m^0(0,k) = N(a_1'a_2'\ldots a_{k-1}')$$

$$= \sum_{j=0}^{k-1} (-1)^j S_j$$

where $S_j = \Sigma N(a_{i_1} a_{i_2} \ldots a_{i_j})$. This formula can be found in any combinatorial book, e.g. chapter 4 of [7].

Stone claimed that $S_j = \binom{k-1}{j}(m-j-1)_{k-j-1}$ and got the solution:

$$C_m^0(0,k) = \sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j}(m-j-1)_{k-j-1} \qquad (2.7)$$

By substituting this into equation (2.3) we can get

$$C_m(j,k) = \binom{k-1}{j} \sum_{n=0}^{k-j-1} (-1)^n \binom{k-j-1}{n}(m-j-n-1)_{k-j-n-1}$$

This is indeed an improvement over Burnett and Coffman's result in terms of computational complexity. However, Stone did not show how he got S_j.

Although Burnett and Coffman got a recursive solution, their equation can be expanded into Stone's result. We show this expansion in Appendix A. This reveals that their results support each other.

Actually, the derivation of S_j is not a trivial problem. In Appendix B, we show a way to derive S_j. Our result shows that S_j is indeed equal to $\binom{k-1}{j}(m-j-1)_{k-j-1}$ and this completes what Stone did not do in his paper.

As a matter of fact, if we know S_j we can just plug it into another inclusion-exclusion formula and solve the original problem, i.e. $C_m(j,k)$, even more directly. The formula is:

$$e_j = S_j - \binom{j+1}{1} S_{j+1} + \binom{j+2}{2} S_{j+2} - \ldots + (-1)^{k-1-j}\binom{k-1}{k-1-j} S_{k-1}$$
$$= \sum_{n=0}^{k-1-j} (-1)^n \binom{j+n}{n} S_{j+n}$$

where e_j is the number of objects that have exactly j properties and S_j is the number of objects that have at least j properties. If we define a property to be that an α-transition must occur at a certain position in a k-length sequence, just as Stone did, then $e_j = C_m(j,k)$ and S_j is what we

proved in Appendix B. So

$$C_m(j,k) = e_j = \sum_{n=0}^{k-1-j} (-1)^n \binom{j+n}{n} \binom{k-1}{j+n} (m-j-n-1)_{k-j-n-1}$$

Notice that all sequences considered here begin with a 0. By a simple manipulation of those binomial coefficients, it is very easy to show the above result is the same as Stone's result,equation (2.7). However, our solution is more direct and doesn't need Burnett and Coffman's theorem.

For the limiting case $\alpha = \beta = 1/m$, Burnett and Coffman's bandwidth equation can be found to have the same numerical value as Hellerman's result.

Although Burnett and Coffman did not give a nice result for their model, they did show a very important phenomenon about the serial correlation between requests. That is, when α increases, the bandwidth will increase exponentially. They claimed that for most programs α is about 0.25, so the bandwidth will be higher than Hellerman's result when $m > 4$.

All we said in this section can be found in [8], and this problem is now finished. Later, Burnett and Coffman had several papers about interleaved memory systems and we will describe them in the next section.

2.3 Burnett and Coffman's Other Models

In [9], Burnett and Coffman continued to consider Hellerman's model in more detail. However, they changed the strategy. They separated the data and the instruction requests and considered their individual bandwidth $DB_{av.}$ and $IB_{av.}$. The resultant bandwidth is the sum of these

two which shows the average number of memory modules in operation on data or on instructions during a memory cycle.

In a real program, we usually put successive instructions in successive memory modules, so there won't be any conflict due to instruction references. But there will be branch instructions in the program which will cause some instruction requests generated during a memory cycle to be wasted. They called this internal waste. In their analysis, they put a parameter λ to denote the probability that a branch will occur during a memory cycle, or the probability that internal waste occurs. Then it is easy to get:

$$IB_{av.}(n_i, \lambda) = \sum_{k=1}^{n_i} k(1-\lambda)^{k-1}\lambda$$

where n_i is the number of instruction requests scanned in one memory cycle.

For data requests, they just applied the result of [5] to be $DB_{av.}$ which is a function of α and n_d, the number of data requests considered in one cycle. Combining $IB_{av.}$ and $DB_{av.}$ yields the total bandwidth. This scheme allows referencing different numbers of instruction and data addresses during one memory cycle. Since the resultant bandwidth is a function of α and λ as well as n_i and n_d, for fixed α and λ we can adjust n_i and n_d to reach the maximum bandwidth.

This paper seems to be closer to the real world. However, control might be a big problem.

Later, Burnett, Coffman and Snowdon presented an interleaved memory model using queue to store blocked requests [10]. They do not stop accessing at the first conflict, instead they just put the blocked requests into a conflict buffer and keep going until either all memory modules are busy or the buffer is full. In the next cycle, the system will scan the

buffer first, then go on to the input stream. They showed the influence of the queue length on the resulting bandwidth.

Their result indicates that the use of a queue will improve the performance. However, they treated the input stream as a sequence of independent module numbers. This is not true in real programs. So the result of this model isn't very practical. But it did give a very good suggestion for a system design.

In this paper, they didn't give an analytical solution since it is very difficult to get due to the appearance of a queue. They used simulation to solve the problem. In Chapter 4 we will show four of our models, two of them have queues in the system, and we also use simulation to get the result.

Recently, they published another paper [11] which uses a technique called Group Request Structure that combines those techniques shown in their last two papers. They used a model with a conflict buffer like that in [10]. However, there are two input streams coming in, one for instruction requests and the other for data requests.

In every memory cycle, they analyzed M_i instruction requests. Just as in [9], they assumed that the instruction requests do not have conflicts except with a branching probability λ. So analytically, instruction requests contribute $\sum_{k=1}^{M_i} k(1-\lambda)^{k-1}\lambda$ to the total bandwidth.

As for data, they used the new model and simulated by using the method of [10]. The input data requests they generated will have a probability α such that an α-transition occurs. In every cycle, at most M_d data requests will be satisfied. Again, when a conflict occurs the later request will be put into the buffer.

The final bandwidth will be the sum of the instruction bandwidth and the data bandwidth, which is a function of M_i, M_d, α, λ, and the queue length L. By changing M_i, M_d and L, they can reach a maximum bandwidth for fixed α and λ.

They claimed that this model has the best performance. However, it is even more artificial due to adding bandwidths.

2.4 Ravi's Model

In a short note [12], C. V. Ravi presented a model of a multiprocessor with an interleaved memory system. He described a system with p processors and m memory modules. In every cycle, each of the p processors will generate a request ranging from 0 to m-1. He predicted the average memory bandwidth by mathematically figuring out the average number of distinct numbers among these p module numbers.

This is a well-known combinatorial problem and the solution is:

$$B_{av.} = \sum_{k=1}^{t} k \frac{k!S(p,k)\binom{m}{k}}{m^p} \qquad (2.8)$$

where $t=\min(m,p)$ and $S(p,k)$ is the Stirling number of second kind. $k!S(p,k)$ is the number of ways to put p distinct objects(requests) into k distinct boxes(modules) with each box containing at least one object. $\binom{m}{k}$ is the number of ways to choose k modules out of m modules.

Ravi showed some performance curves for his model. But he did not plot his bandwidth curves against fixed ratios of m and p, and that is something interesting he missed.

Actually, equation (2.8) can be reduced to a very simple closed form, that is:

$$B_{av.} = m\left[1-(1-\frac{1}{m})^p\right]$$

We show this derivation in Appendix C. If we plot this result against p for fixed ratios $r = {}^{p}/_{m}$, we get a family of curves as shown in Figure 3. As you can see, all these curves are almost linear. This is not surprising since you can further reduce the above equation to an asymptotic form by using the fact

$$\lim_{p \to \infty} (1 - \frac{1}{m})^{p} = \frac{1}{e^{r}}$$

We repeat the derivation of this limit value in Appendix D. So the asymptotic equation is:

$$\begin{aligned} B_{av.} &= m(1 - \frac{1}{e^{r}}) \\ &= \alpha(r)\, m \\ &= \frac{p}{r}(1 - \frac{1}{e^{r}}) \\ &= \beta(r)\, p \end{aligned}$$

which is a linear function of either m or p. Obviously, this result is quite different from what we would expect by using Hellerman's square root concept. In Chapter 4, we will show that our models have the same result.

Of course, you might say that Ravi used p processors rather than 1 and that's why he got linear result. But Hellerman assumed that there are always more than m requests available, or equivalently, the processor can generate at least m requests per memory cycle. So we might compare Ravi's result against Hellerman's result with M=p.

In order to compare them, we plot $B_{av.}$ against m curve of Ravi's model for a certain p in Figure 4, superimposed with Hellerman's result when M=p. Both of them are analytical solutions.

Two things can be seen from this diagram; first, Ravi's bandwidth is exponentially increasing and approaches p when m gets large, and second,

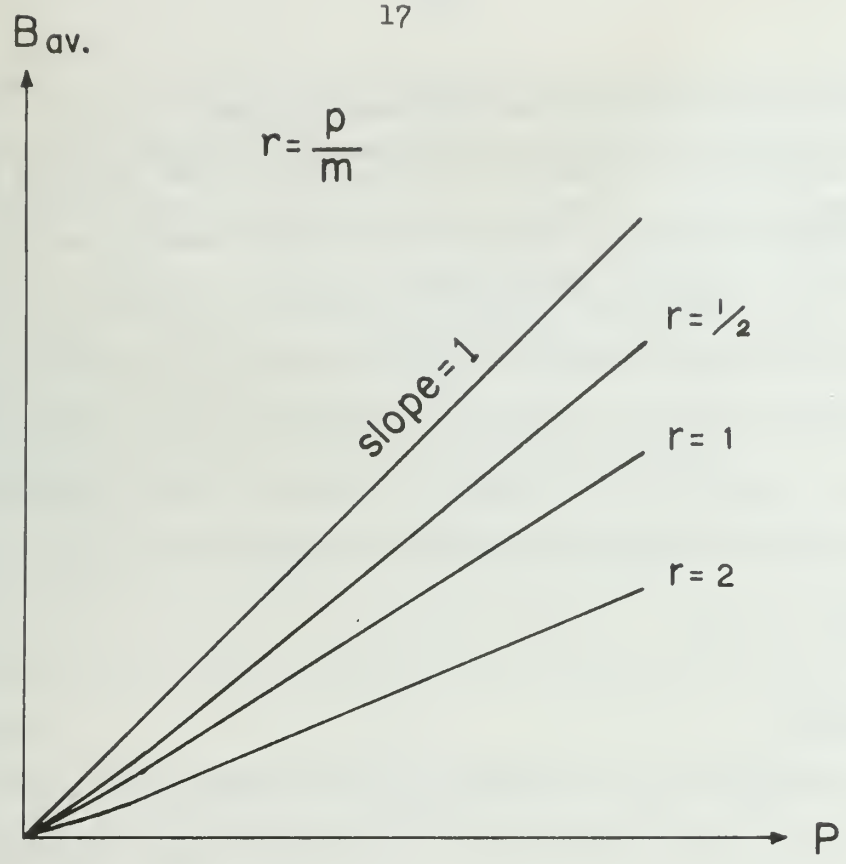


Figure 3. Bandwidth Curves of Ravi's Model for some Fixed Ratios of p and m

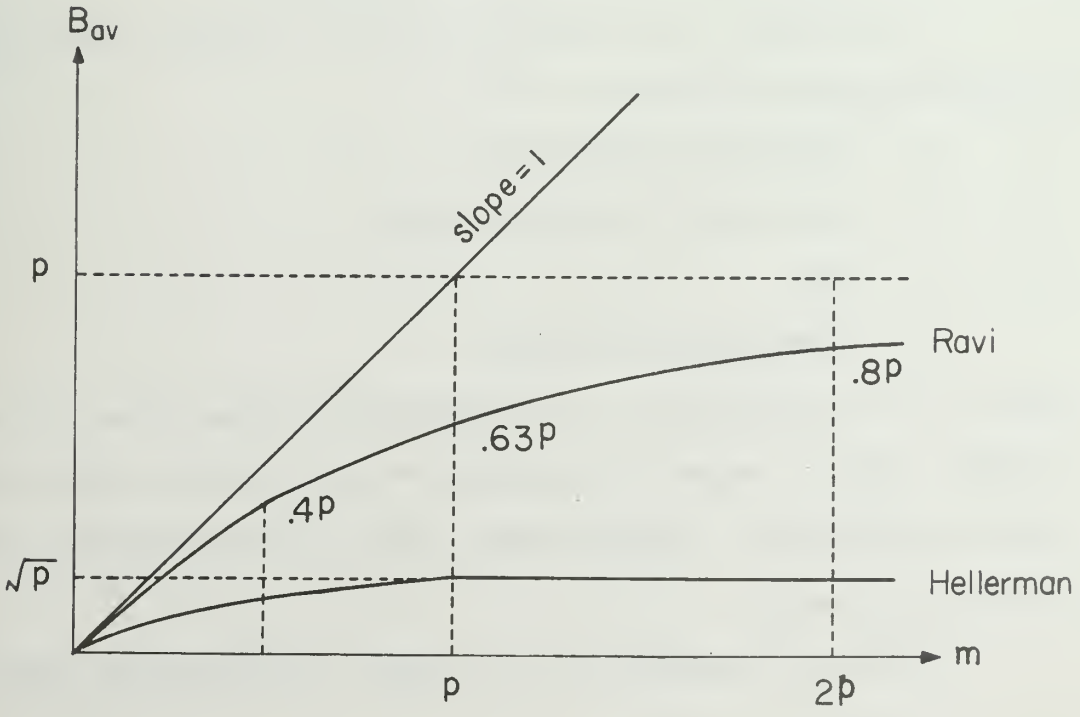


Figure 4. Comparison Between Ravi's Model and Hellerman's Model

Ravi's curve is much higher than Hellerman's curve.

Although Ravi claimed that his model allows queueing in the memory modules, he did not explain how to build these queues and what the impact of these queues is. Besides, he failed to explain how to handle the conflicts and what happens to those blocked requests that can not be satisfied in the present cycle, i.e. he ignored them. But his paper did give us some inspiration in building our models, and the linearity shown in Figure 3 indeed triggers us to find out whether it is true for all other models or not.

2.5 Conclusion

All the models introduced in this chapter have a common flaw, that is, none of them mentioned the data dependency possibly exists between the requests. Thus they give people an illusion that their models can be fitted into any machine. However, this is not true.

For example, let us take a look at the following piece of assembly code which performs the task A = B + C:

a LOAD B INTO ACCUMULATOR

b LOAD C INTO AUXILIARY REGISTER

c ADD

d STORE THE RESULT INTO A

where a, b, c, d are assumed to be the addresses of these four instructions. Certainly, when we execute this sequence, we can not execute (but can fetch) d until c is finished. In other words, there is a data dependency between these two instructions.

Using our terminology, we will have an input request stream like:

a, B, b, C, c, d, A

If we have a very fast and fancy processor, probably using a look-ahead scheme, we can generate all these seven addresses in the same memory cycle. Even though all these requests refer to different memory modules, i.e. there is no conflict, we can't execute STORE A due to the data dependency between c and A. So when we feed this string into any model discussed in this chapter, the real result is not what they would expect. By "real" result we mean if we run the program correctly.

There are also some other kinds of data dependency. The data dependency will greatly constrain the usage of a model. Although nothing is wrong with all these analyses, we must keep in mind that no model is "universal".

So before we go on to our models, we will study some data dependency problems. Then we can use the result to justify the usefulness of all the models, including ours.

3. DATA DEPENDENCY

3.1 Data Dependency Types

As far as the memory design is concerned, the data dependencies in real machines can possibly be classified into four different types. We depict them graphically in Figure 5. A small circle represents a request (or an address) and an arrow indicates the dependency relation. For example, in Figure 5 there is an arrow from a to b, this means request a can not be serviced until request b has been satisfied. A horizontal string of circles means those requests that will be investigated at the same time. They might be generated either by seperated processors or by a fast processor. A vertical string of requests always consists of requests generated by the same processor.

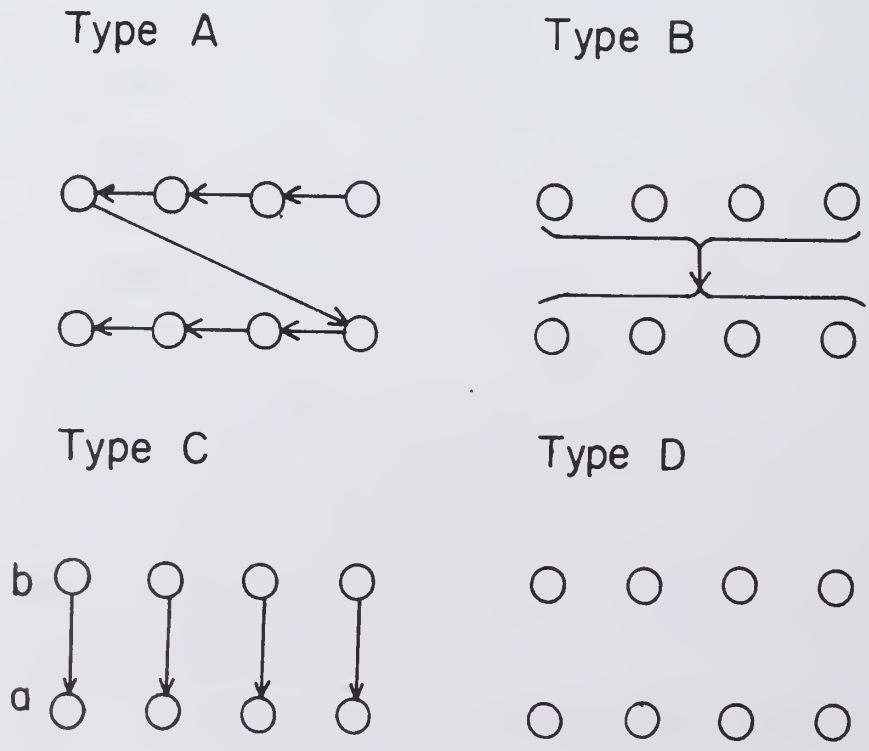


Figure 5. Four Types of Data Dependency

Type A is the only one with horizontal data dependency. That means there are data dependent relations between the requests scanned at the same time. This is the worst case we can have since we must do everything one by one. The horizontal arrows might not appear all at the same time, here we only show all the possibilities by drawing all the arrows explicitly.

Usually, type A occurs when the machine is running a serial program. Of course, we can draw the picture as a single vertical string. But we fold the string in order to cover the case when several processors are running one program. All standard serial machines fall into this category. Here is a FORTRAN example of type A data dependency:

```
      DO 10 J = 1, P*H
   10 M(I(J)) = M(J) + C
```

All assignment statements must be executed serially.

Type B data dependency is the one when some (or all) of the requests at second level have data dependent relations with some (or all) of the requests at first level. The relations might not be one to one and might change every cycle. So in this case, the first level must have all been done before the second level can be served. In Figure 5, we represent this by showing only one vertical arrow.

This type of requests usually arise in a vector machine, such as CDC STAR. The first vector must be completely done before the second vector can be launched. For example, if we have a matrix M stored in p memory modules in a standard way, i.e. a column will be in one module, then do the following double DO loops:

```
      DO 10 K = 2, H
      DO 10 J = 1, P
   10 M(I(K),II(J)) = M(K,J)
```

Since the subscripts at the left-hand side are variables, the best we can do is to proceed row after row. After we fetch a row, we must store all p elements into memory before we can fetch another row. This is exactly the type B data dependency. You can find a lot of type B data dependency in the differential equation problems.

For type C, there is no arrow across the vertical boundaries. The data dependent relations only occur in vertical strings. All requests along a vertical string must be done serially. This type happens when several processors are running seperate tasks (or programs). Some multiprocessor machines are operated in this fashion.

Here is a FORTRAN example with type C data dependency:

```
      DO 10 K = 1, P
      DO 10 J = 1, H
   10 M(K,I(J)) = M(K,J)
```

It is easy to see that the permutation in a row must be done one by one, but all rows can be operated simultaneously.

The last type is the one with no data dependency relation anywhere. All the requests are generated independently. Although this type is not found very often in real machines, it is possible. It is easy to analyze and has been used in many previous models.

This usually occurs at the beginning of a program when we initialize the program. For example:

```
      T = 0
      DO 10 I = 1, 100
      A(I) = 1
      DO 10 J = 1, 50
   10 B(I,J) = 0
```

There is no connection between any two variables, so we can initialize them randomly. One example of this type of data dependency in the memory for programs with more data dependency is the IBM 360/91, where the independency between requests may be made by using Tomasulo's algorithm in the processor, queues and tags.

All cases in the real machines can be classified into these four types, and they might be used to justify the usefulness of a model.

3.2 Summary

Table 1 below shows the machine type and interleaved memory models most suitable for a certain type of data dependency. We have already explained the second column in the last section. Now we are going to explain how we fit Hellerman's, Burnett, Coffman and Snowdon's and Ravi's models into third column.

If you recall, Hellerman's model stops at the first conflict and assumes no data dependency up to that point. The only explanation of this operation is that all the incoming requests are independent so it can keep going until a conflict is met, and the control unit of this model is so simple that it doesn't know how to handle the conflict, hence it stops. So we fit it into type D according to the data dependency.

Snowdon, Burnett and Coffman's model also ignores the data dependency. It throws the blocked requests into a conflict buffer for later processing and keeps looking for more requests. So their model is also classified as type D.

As for Ravi's model, all p processors will generate a new request independently in every cycle. So we fit it into type C. However, he did not explain how to handle the conflicts. Apparently, he just throws those blocked requests away since there is no queue provided in his model. This is a big flaw. But here we are only concerned with the data dependency type, so we place it in type C group.

In the next chapter, we will present four interleaved memory models. Primarily, we design them for solving different types of data dependency. We also put them in Table 1 for later reference. The reason will become apparent after we describe them.

Data Dependency Type	Machine Type	Interleaved Memory Models
A	Old standard serial machines	None
B	Vector machines, e.g. CDC STAR	I, II
C	Standard multiprocessor machines executing independent tasks	Ravi, III
D	Tomasulo type machines, e.g. IBM 360/91	Hellerman, Coffman, IV

Table 1. Data Dependency Types with Their Associated Machine Type and Most Suitable Interleaved Memory Models

4. OUR MODELS AND RESULTS

4.1 Introduction

All the models described in Chapter 2 are based on one assumption, that is, the system has an infinite supply of requests and processors never run out of requests. In other words, the system is always in a steady state. This is certainly not always true in real machines. In our analysis, we also consider the case when the request supply is finite and we call this a transient state. In a transient state, some fringe effects will occur and we are going to show how this fringe effect influences the performance of the system.

In this chapter, we will present four interleaved memory systems we have been working on. The approach we use is more realistic than others. Both steady state and transient state performances will be analyzed. One thing that nobody else has worried about before, viz. how to queue the memory conflicts, will be considered here. This is an important factor which will greatly influence the design and use of a model.

All our models are multiprocessor and multimemory systems, i.e. each has p processors and m memory modules. The basic difference between these four models is the way they handle requests. As we said in the last chapter, each model can only fit into a certain class of machines. For example, Model I and Model II will satisfy all requests generated in one cycle before going on, so they can be implemented in a vector machine like CDC STAR where all processors are running the same program and data dependency is important. Model III and Model IV allow every processor to generate a new request in every cycle if possible, and the blocked requests

will be stored in the waiting queues. So they can only be used in a MIMD machine, such as B6700, where each processor is running an independent job, or in some special cases on a SIMD machine, or in a vector machine where vectors have tags.

The reason we limit our models to a certain class of machines is to fit the type of data dependency between requests. We have pointed out this in the last chapter. We are not proposing a best way of designing interleaved memory systems here. Instead we are trying to tackle this problem in a more realistic way in order to give the flavor of how to design a proper system for a particular machine.

In the following sections, we will first give a description of each model, followed by an analytical or simulation result. Then the comparisons of their performances and costs will be given in the next chapter which can be used as a design guide.

In all our models, we will assume that all memory modules operate synchronously and with identical cycle time. The requests generated by the processors will be the memory module numbers instead of conventional storage addresses. The definition of bandwidth we are going to use is slightly different, we will divide the total number of requests satisfied by the total number of memory cycles required. So we are viewing things from a macroscopic point of view. But essentially, this is the same as the other definitions.

4.2 Model I: Request Slicing

Figure 6 shows the logic structure of our model I. There are p processors and m memory modules. Single lines represent the control flows and double lines represent the data flows.

At the beginning of a cycle, each processor will generate a request which is an integer from the set {0, 1,..., m-1} . All p requests will be sent into a conflict resolution box which handles the memory conflicts, which occur when two or more processors generate the same module number. Those requests that will be served then go to a switching network, such as a cross-bar alignment network, which will switch them to the proper memory modules. Those who are blocked will be reprocessed in the next memory cycle. After all these p requests have been satisfied, the processors are allowed to generate a new burst of requests. This mode of operation is the way of handling type B data dependency. So this model can be used in type B machines.

The details of how the conflict resolution box works can be seen in Chapter 5 when we give a design for it. If it is a read operation, the data fetched from the memory must also be switched back to the processors. So the switching network must be bi-directional or have two copies. Notice that on the way back there is no conflict problem, so the returning infomation can be sent to the switching network directly.

Let's take an example to see how this model operates. Suppose we have 8 processors and 8 memory modules. The 8 requests generated are:

3, 4, 0, 1, 3, 4, 7, 4

These requests can be viewed as a distribution histogram shown in Figure 7. We may slice them into 3 pieces and satisfy them in 3 memory cycles. This is why we call this model a "request slicer".

Obviously, the bandwidth of this model depends on the maximum height of the request distribution. In the above case, the maximum height is 3 so we must spend 3 memory cycles to access 8 words. Consequently, the

bandwidth (in words/cycle) can be defined as 8/3 = 2.33. So the performance, or the average bandwidth of the model, can be found by figuring out the average height of the request distribution.

This is an interesting combinatorial problem. We derived an analytical solution for the average height shown as equation (4.1), and we will explain every term in this expression. The average height is:

$$H_{av.} = \sum_{h=1}^{p} h\, P(h)$$

$$= \sum_{h=1}^{p} h \frac{\sum_{j=1}^{\lfloor p/h \rfloor} \left(\prod_{k=0}^{j-1} \binom{p-k*h}{h} \right) \binom{m}{j} f_{p-j*h}(m-j,h-1)}{m^p} \qquad (4.1)$$

The function P(h) is the probability that the maximum height of the distribution of p requests is h. We multiply it by h and sum over all possible h's which gives the average height of the distribution $H_{av.}$. The expression in the numerator of P(h) is the total number of sequences such that at least one module number occurs exactly h times and other numbers occur no more than h times. Denominator m^p is the number of all possible sequences with repetitions. The ratio gives the probability function. Notice that there might be more than one number that occurs h times, that is why there is a summation in the numerator. The maximum number of such numbers (each occurs exactly h times) is $\lfloor p/h \rfloor$ which serves as the upper limit of the summation. The product of the binomial coefficients is the total number of ways to choose j*h positions out of p positions so as to distribute those j numbers such that each occurs exactly h times, or equivalently, assign j different requests to j*h processors so that every h of them have the same request. $\binom{m}{j}$ chooses j numbers from m possible numbers. And $f_{p-j*h}(m-j,h-1)$ takes care of the rest of the positions. Hence the numerator indeed covers all the possible cases.

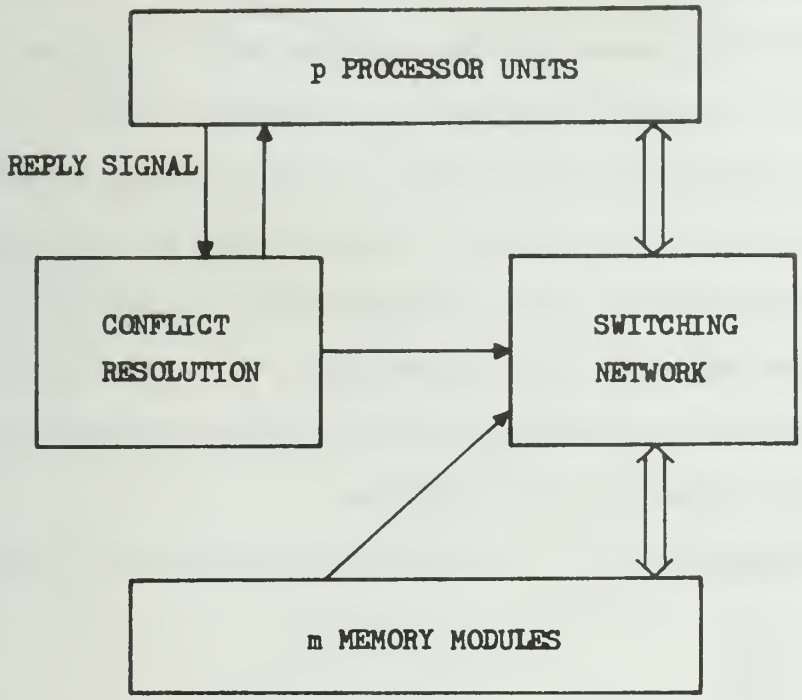


Figure 6. Block Diagram of Model I

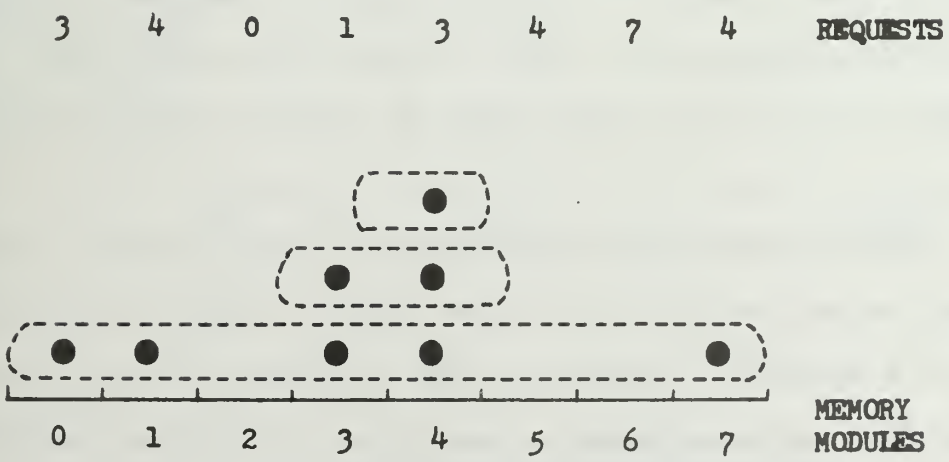


Figure 7. An Example of Request Slicing

The function $f_n(m,s)$ is the number of ways to distribute n distinct objects into m distinct boxes such that each box contains at most s objects. Here the object is a processor and the box is a memory module. We use it in equation (4.1) to count the number of ways to place with repetition the remaining m-j numbers into the remaining p-j*h positions with the restriction that the maximum occurrence is h-1, or equivalently, assign requests to the remaining p-j*h processors so that no more than h-1 of them have the same request. The solution of this function can be found in Chapter 4 of [13] and we repeat it in Appendix E for reference.

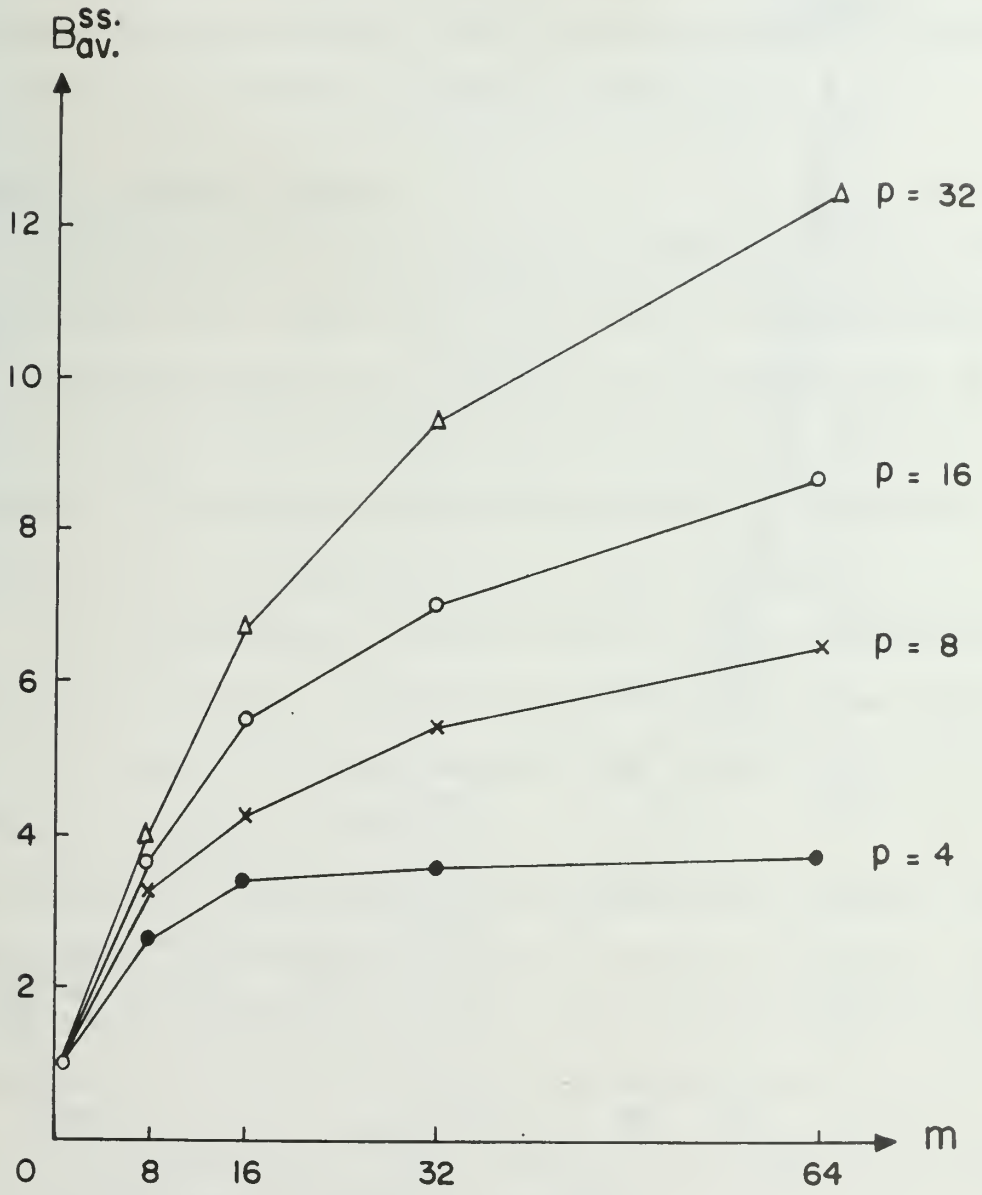
After finding out $H_{av.}$, we can calculate the average bandwidth by

$$B_{av.}^{ss.} = \frac{p}{H_{av.}}$$

as we explained at the end of the last section. Since the analytical result is essentially the steady state result, we put a superscript ss. to denote this fact.

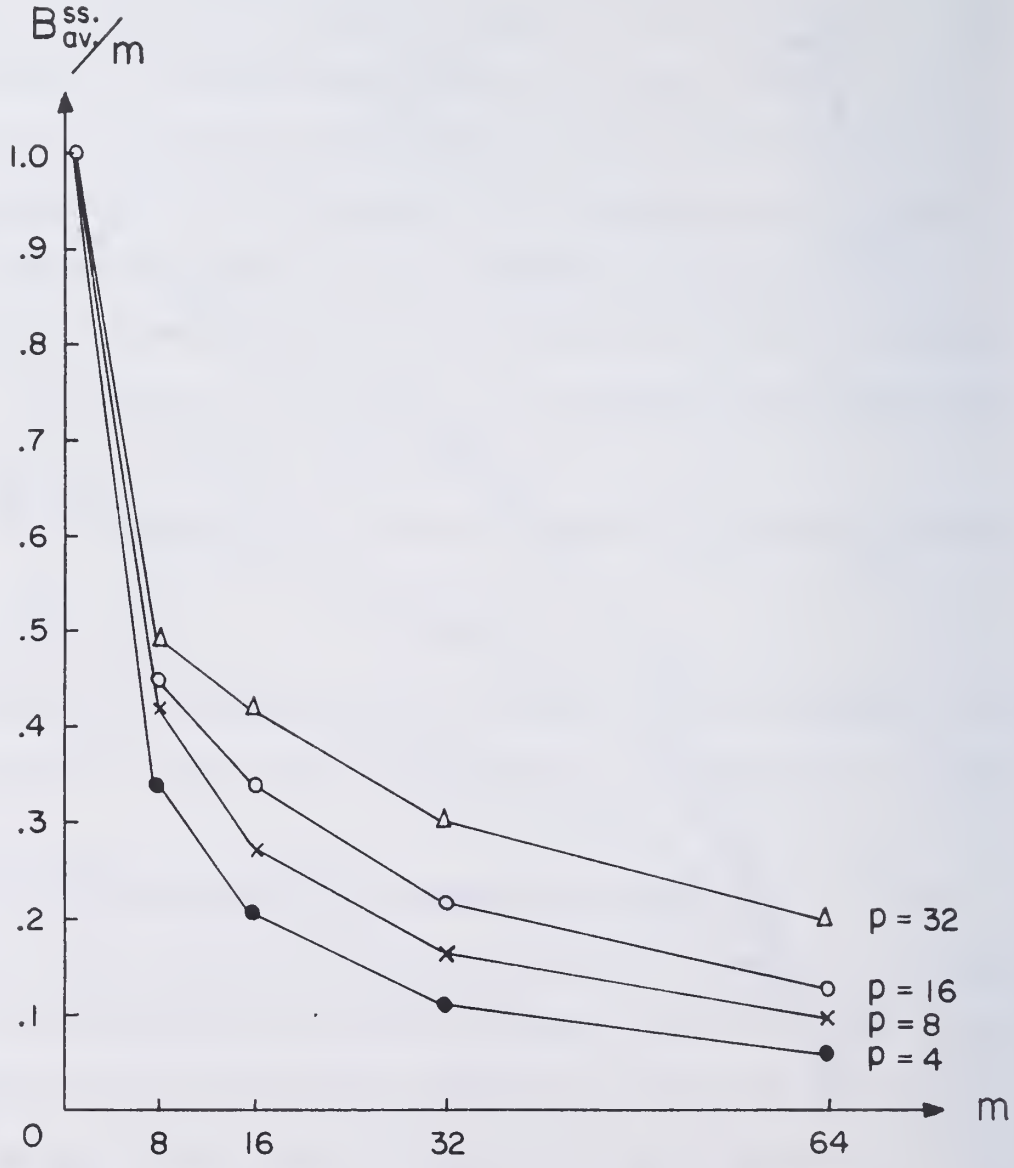
Figure 8(a) shows $B_{av.}^{ss.}$ versus m curves for various p values. Apparently, we can increase the bandwidth either by increasing the number of processors or by increasing the number of memory modules or by both. From this diagram, we can see the former effect is slightly greater than the later.

As Ravi argued in his paper, Hellerman's model assumes that there are at least as many requests in the input stream as there are memory modules and when p = m his model is much better than Hellerman's. As you can see from Figure 8(a), our model shows the same thing, although the bandwidth is not as good as Ravi's. However, our model I is more realistic.



(a) Steady State Bandwidth Curves

Figure 8. Bandwidth Curves of Model I



(b) Normalized Bandwidth Curves

Figure 8 (continued). Bandwidth Curves of Model I

Figure 8(b) shows the normalized bandwidth $B_{av.}^{ss.}/m$ plotted against m. This graph shows the utilization of the memory modules. When p = m, the utilization is quite low. This is what you would expect if new requests can not be generated until the old ones have all been served.

4.3 Model II: Conflict Blocking

One other possible way to handle the conflicts is just to block all those requests that cause conflict and satisfy the others in the present cycle. All blocked requests will be piled up in a separated area, and will be served one by one in the following cycles. After all p requests have been satisfied, the processors will generate another set of requests. This is the second model we analyzed and it also belongs to type B.

The logic structure is shown in Figure 9. Here we substitute the conflict resolution box by a conflict detection box which is a combinational circuit that signals a 1 to those who cause conflict. The design of this box is simple but costs a lot of gates.

If we use the same example in the last section, we need 6 memory cycles to satisfy these requests, hence the bandwidth is reduced by one-half. The order in which these requests will be served is shown in Figure 10.

We also have an analytical solution for this model. The average number of cycles needed to satisfy p requests is:

$$C_{av.} = \frac{(m)_p}{m^p} + \sum_{c=3}^{p} c \, \frac{\binom{p}{p-c+1} (m)_{p-c+1} \left(\sum_{j=1}^{\lfloor (c-1)/2 \rfloor} \binom{m-p+c-1}{j} g_{c-1}(j,2) \right)}{m^p}$$

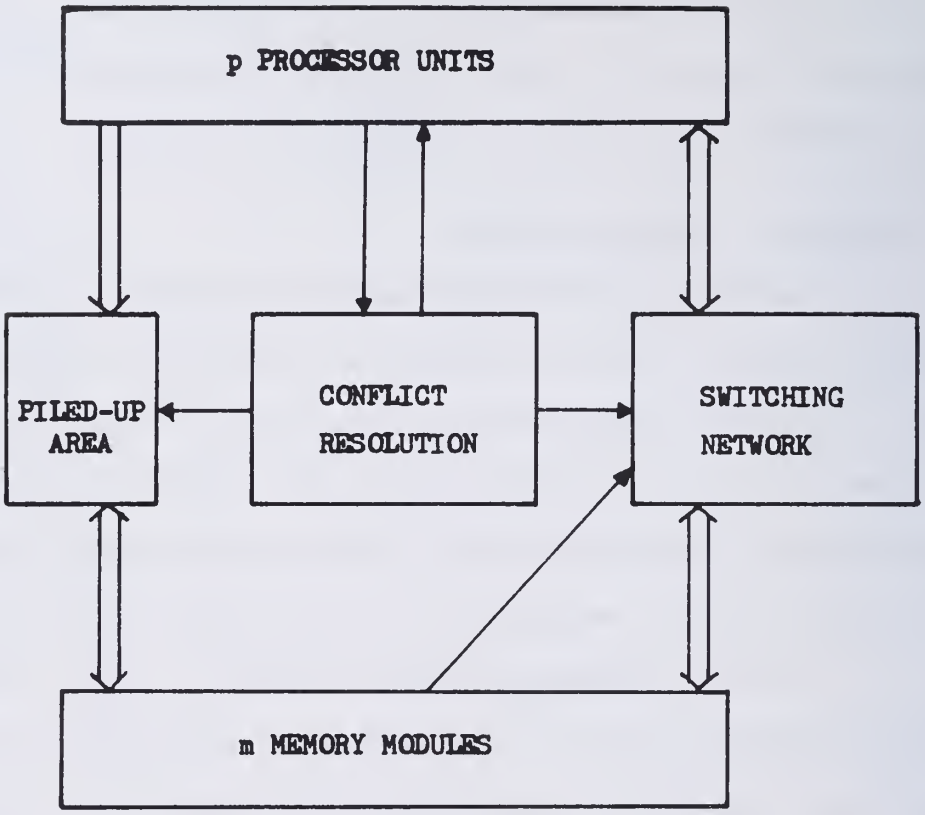


Figure 9. Block Diagram of Model II

3	4	0	1	3	4	7	4	REQUESTS
(2)	(3)	(1)	(1)	(4)	(5)	(1)	(6)	HONORED ORDER

Figure 10. The Service Order of the Same Example by Using Model II

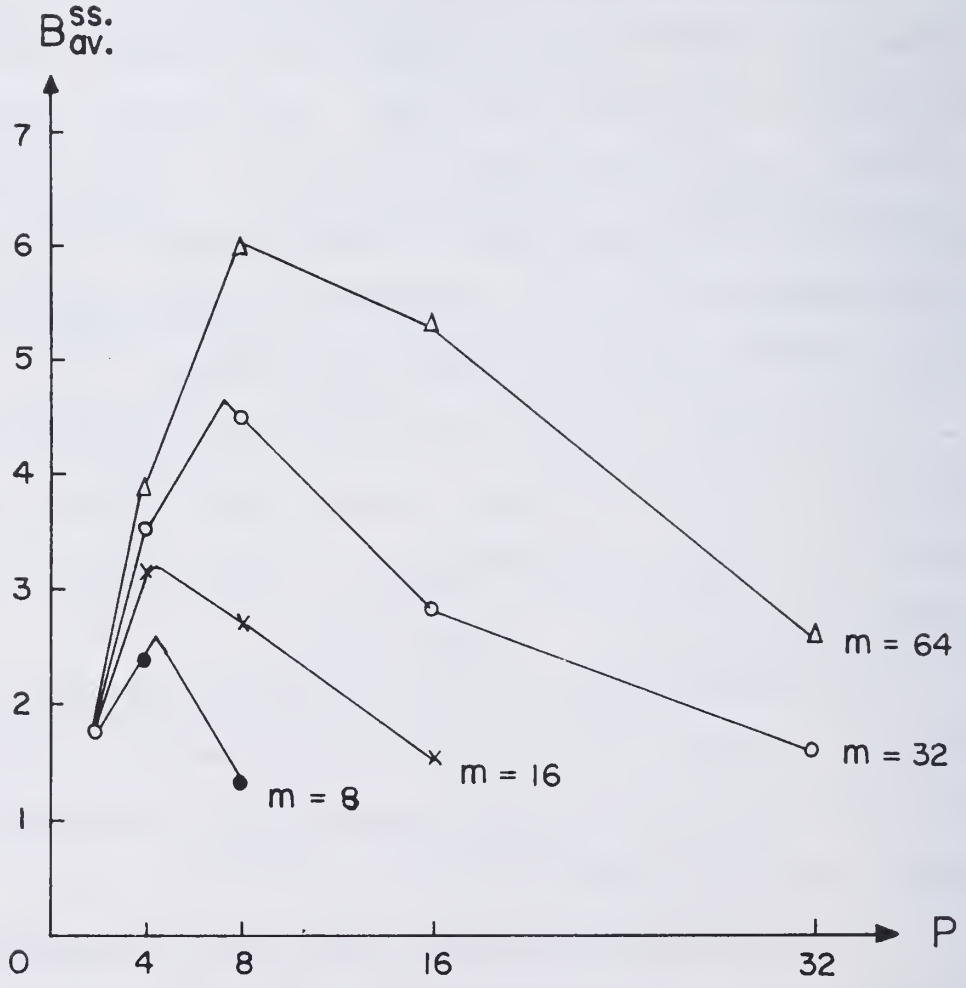
$(m)_p$ is the number os sequences with p distinct numbers that can be satisfied in only one cycle. Since no sequences will be satisfied in two cycles when $m > 1$, the summation goes from 3 to p. $\binom{p}{p-c+1}(m)_{p-c+1}$ is the number of ways to choose p-c+1 distinct numbers to be the p-c+1 requests that have no conflict and can be satisfied in first cycle. The summation in the numerator is the total number of ways to select j numbers out of the remaining m-p+c-1 numbers and distribute with repetition these j numbers into the remaining c-1 positions with each number occurring at least twice. The upper limit $\lfloor c-1/2 \rfloor$ is the maximum possible number of such events.

The function $g_n(m,s)$ is the number of ways to distribute n distinct objects into m distinct boxes where each box contains at least s objects. In our case, $s = 2$. Again, the solution can be found in Chapter 4 of [13] and we repeat it in Appendix F.

The bandwidth $B_{av.}^{ss.}$ is equal to p divided by the average number of cycles $C_{av.}$. In Figure 11, we show the bandwidth and the normalized bandwidth of our model II. Both of these quantities are much smaller than that of Model I as we would expect.

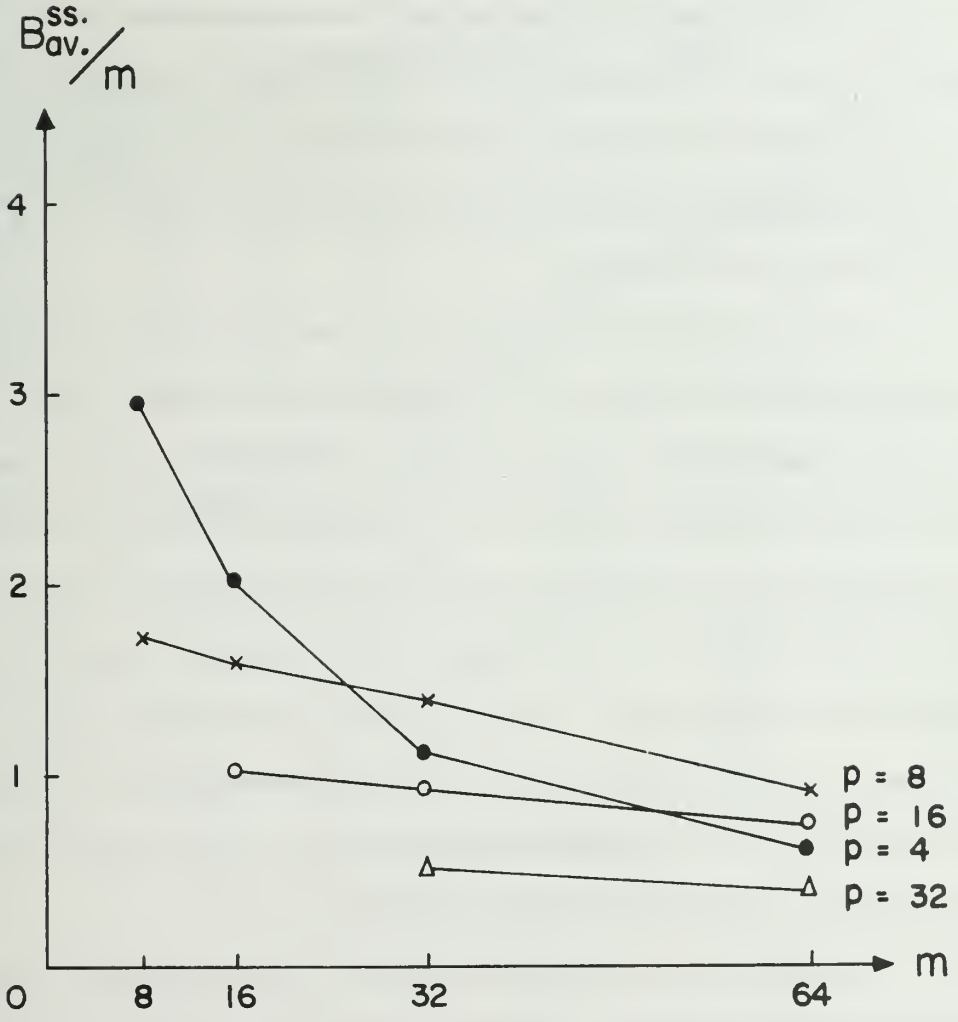
One interesting thing is that when we plot the bandwidth curves against p they show maximum values at $p \cong \sqrt{m}$. This is shown in Figure 11(a). In other words, if you want to use Model II, you had better use p^2 memory modules, or use $\sqrt{m}$ processor units.

Since the bandwidth and utilization of Model II are much worse than Model I and the cost is often higher, we will discard this model. Hence in Chapter 6, we will not show the design of this model.



(a) Steady State Bandwidth Curves

Figure 11. Bandwidth Curves of Model II



(b) Normalized Bandwidth Curves

Figure 11 (continued). Bandwidth Curves of Model II

4.4 Model III: Queueing in the Processor Units

In Models I and II, we did not use any queueing technique. Since Burnett and Coffman have shown better results will be obtained if queues are used, we construct two other models which use queues in two different places to store blocked requests. However, their use will be different.

Figure 12 shows the logical structure of our model III. Basically, the structure is very similar to model I except we build a queue in each processor unit.

At the beginning of each cycle, every processor will generate a new request into the queue if the processor queue is not full. Then all the first elements of the queues will be processed. Again, these requests go into a conflict resolution box. If there is a conflict, the circuit will choose one to be honored according to some criterion, e.g. leftmost first, random selection, round robin, etc.. Rejected requests will remain at the heads of the queues. Those who are selected will be sent into the switching network and dispatched to the address registers of the proper memory modules, and all other requests waiting in the queues will move one place forward. Then another cycle begins.

Figure 13 shows some history of a 4 processor and 4 memory module system with queue length 3. COUNT records how many requests have already been generated by the processor. The circled numbers are the newly generated requests. We see in the second cycle, P_3 does not generate a new request since the queue is full. Here we use the leftmost first algorithm.

It is easy to see this model is very suitable for handling type C data dependency. As a matter of fact, this model is designed for type C programs.

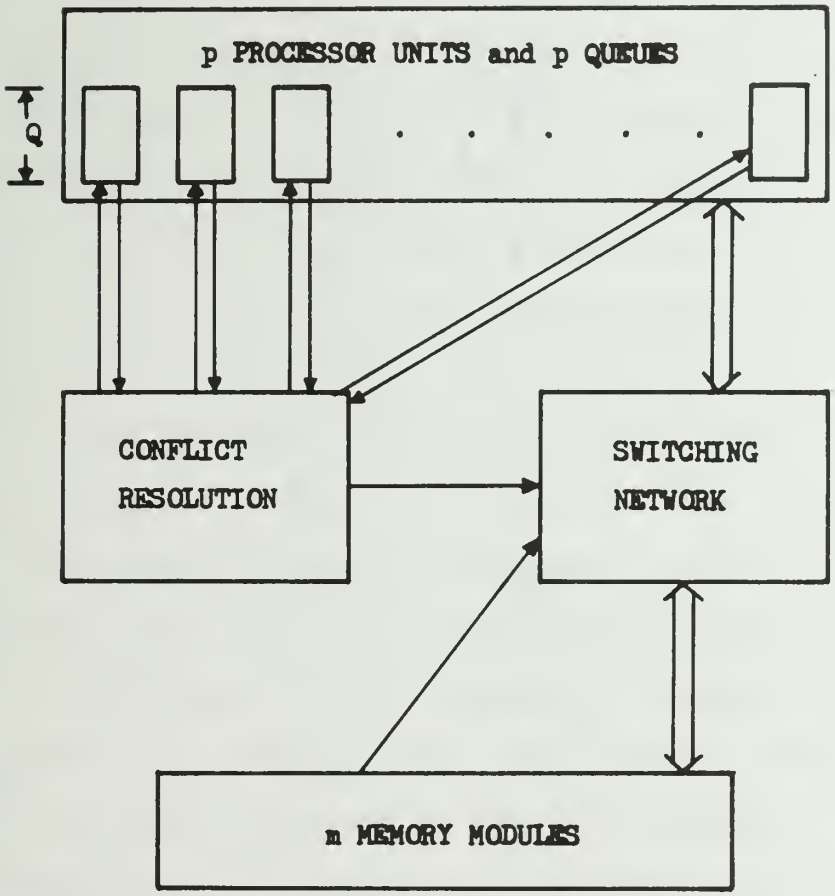


Figure 12. Block Diagram of Model III

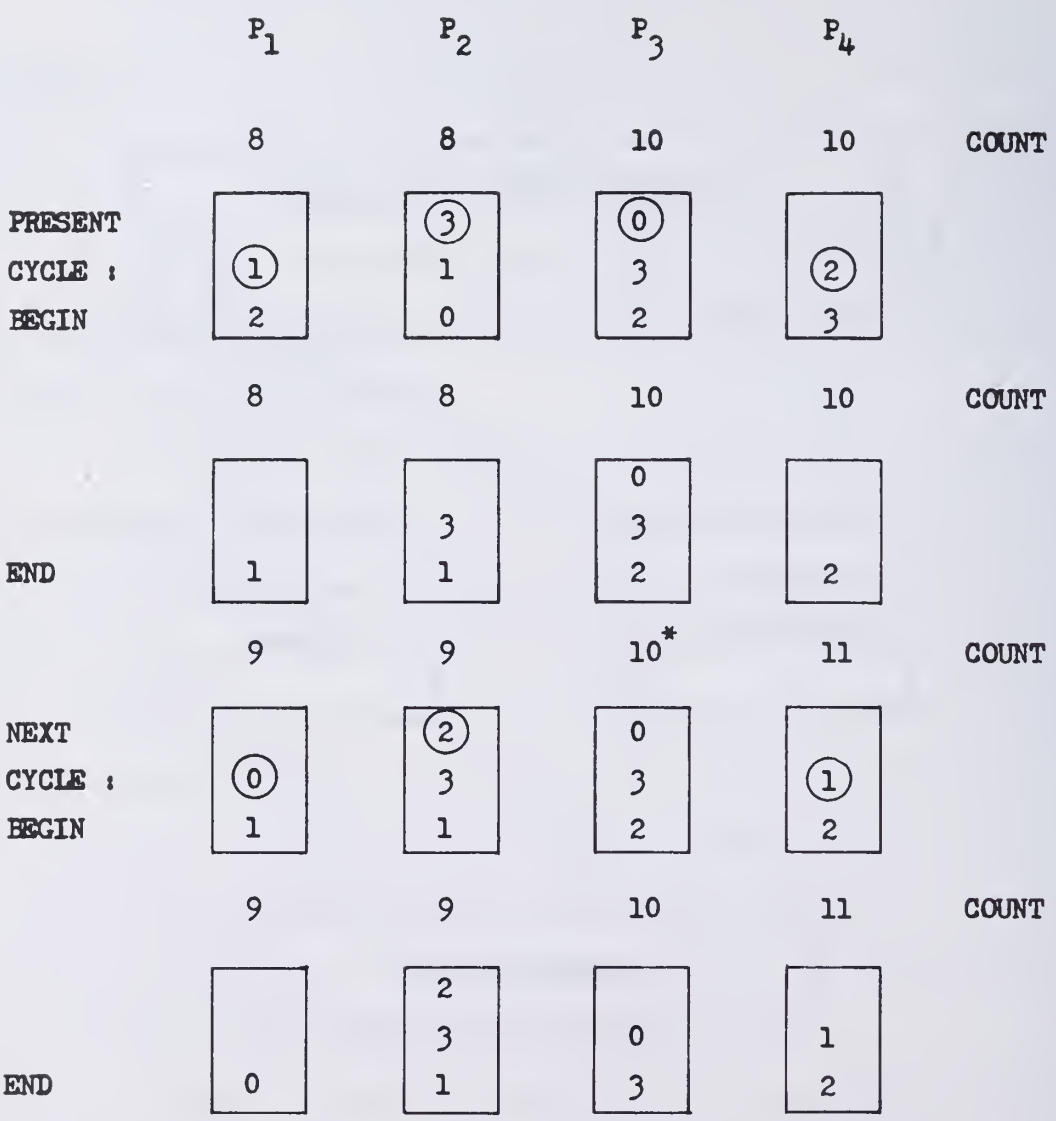


Figure 13. An Example of the History of Processor Queues in A 4-Processor System

Since an analytical solution for this model is very difficult to get, we used Monte Carlo Methods to simulate it. The random number gererator we used to generate the requests is the IBM random number generator RANDU (see Appendix G) with the initial value suggested by [14].

The first simulation result of model III is given in Figure 14, which shows the transient state bandwidth $B^{T.}_{av.}$ versus H, the maximum number of requests each processor generates, when p = m. The transient state bandwidth is defined as the total number of requests, i.e. H*p, divided by the time to finish the whole process. For other ratios of p and m, the shape of the curves remain the same.

The transient state bandwidth is lower than the steady state bandwidth. Because every processor will only generate H requests, as soon as some of the processors stop generating requests the bandwidth of the whole system will start going down. After all processors have stopped, there are still some requests left in the queues and a few more cycles are needed to drain the queues. When we average the whole thing, this fringe effect at the end will lower the resulting bandwidth.

Figure 15 shows the histograms for four different H values. B is the number of requests that are satisfied in a certain cycle, or we may call it the "instantaneous bandwidth". From this figure, we can get a rough idea of how the requests are satisfied. The average bandwidths are marked by X's. If we connect these points together, we will get a curve similar to those shown in Figure 14. So Figure 15 also shows the formation of Figure 14. As H gets larger, the bandwidth approaches the steady state value since the fringe effect becomes smaller.

We will not show the normalized bandwidth (or utilization) curves

here since they are proportional to the bandwidth curves. Better bandwidth, of course, means better utilization.

One interesting observation of Figure 15 is that the area under the histogram is actually equal to H*p, so the shape of the curve will influence the transient state bandwidth. The tail part of the curve is indeed the reason that the transient bandwidth is less than the steady state bandwidth. Thus, if we can manage the queues so that the histogram will become more rectangular, or the tail part becomes smaller, then we will get better bandwidth.

We have done three experiments, each used a different conflict resolution algorithm and they showed very interesting results. The first algorithm is always to choose the request from the leftmost processor if several requests refer to the same module. The second one is to choose a request randomly. The third one is to choose the one generated by the least used processor, or the processor with the smallest COUNT. The results are shown in Figure 16.

From Figure 16, we can see the first algorithm is the worst, the second one has 15% improvement over the first one and the third one is the best with 25% improvement over the first one. So, the fancier conflict resolution circuit we use the better performance we get and the faster the job can be done. Figures 14 and 15 use the first algorithm. We will get the same kind of results if we use other algorithms.

The design of a conflict resolution circuit is not a trivial problem. The first one is relatively easy to implement and we show an example in Chapter 6. If we make some modification, we will get a random selection circuit. However the conflict resolution circuit for the last scheme is

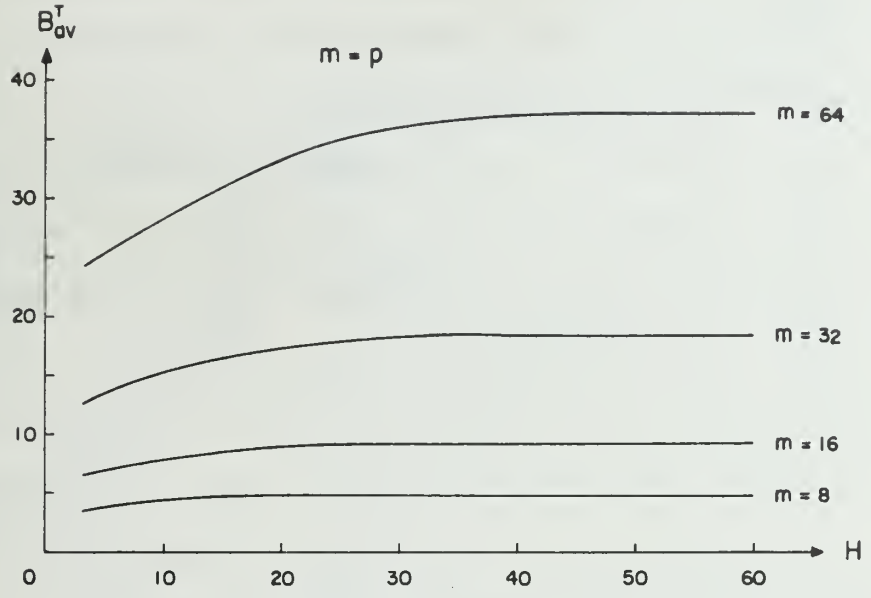


Figure 14. Transient State Bandwidth Curves of Model III when m = p

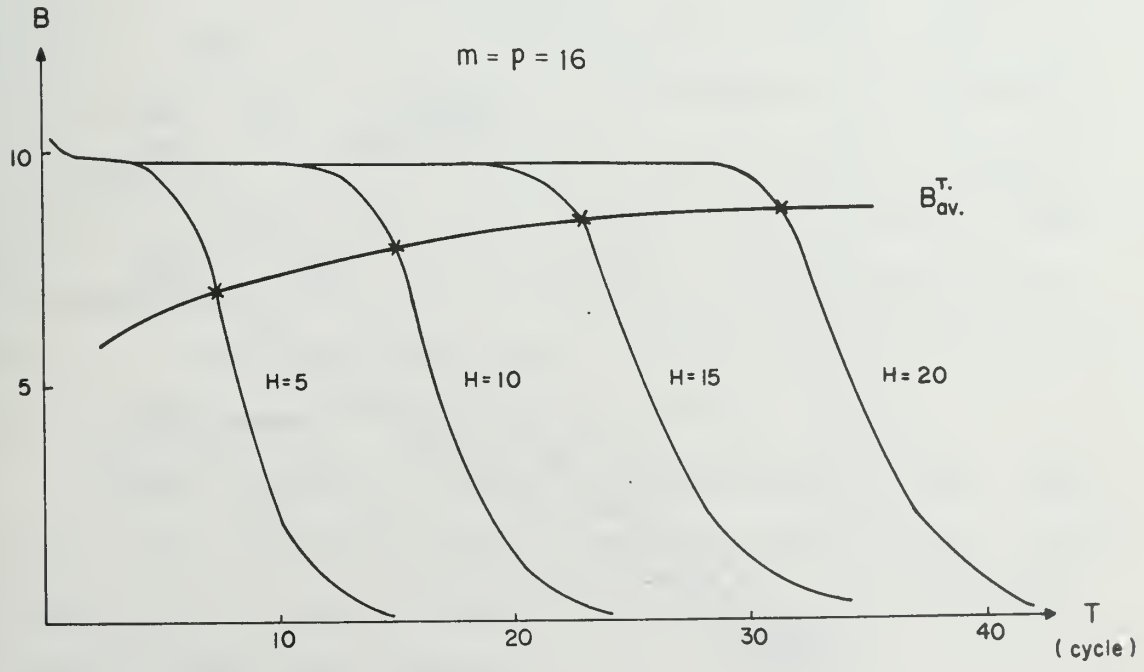


Figure 15. Formation of a Transient State Bandwidth Curve

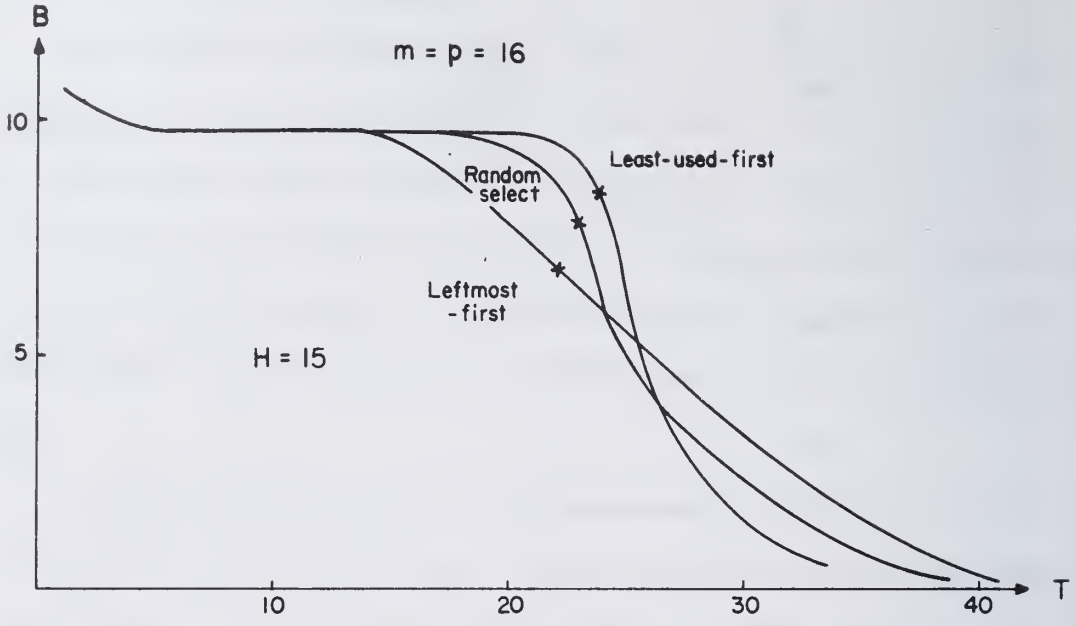


Figure 16. Histograms for Three Different Resolution Algorithms

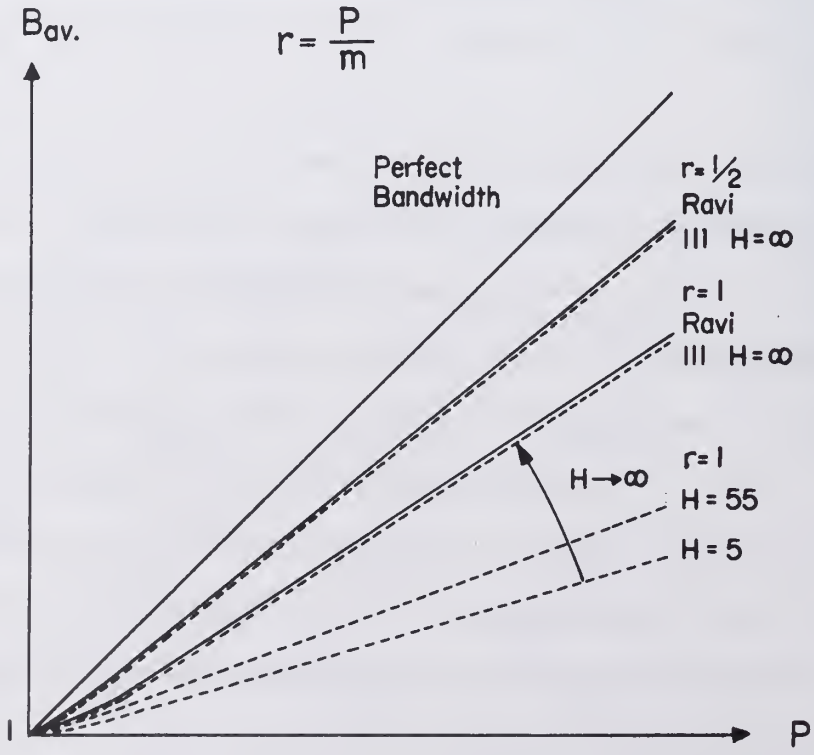


Figure 17. Bandwidth Curves of Model III

prohibitive since it needs a lot of sorting or comparison circuits. So our model III provides a designer the freedom to choose his own conflict resolution circuit according to the trade-off between cost and performance.

Figure 17 shows the steady state bandwidth $B_{av.}^{ss.}$ for different ratios of m and p, together with Ravi's result. The reason we plot bandwidth against p is to show that when we fix p and increase m, the bandwidth will increase and the curve will move upward. Besides, the perfect bandwidth is a slope 1 line in this graph. The $B_{av.}^{ss.}$ curves show the linearity with respect to fixed r. In fact, the transient state bandwidth curves are also linear when r is fixed. Here we show two cases, namely, r = 1 and H = 5, 55, in this graph.

We can see the steady state bandwidths of our model III are essentially the same as Ravi's. Let us explain the reason by two cases. Suppose we have eight processors and eight memory modules, and in the last cycle, three requests were blocked and remained in the queues as shown in Figure 18(a). Suppose the newly generated requests are those shown in Figure 18(b), then Ravi's model will give you bandwidth 7 and our model will only give you 5. However, if the new requests are those in Figure 18(c), then our model will give you 7 but Ravi's only gives you 5. These two cases are equally likely. If a large number of trials have been collected, or the machine runs for a long time, then the outcomes that favor us should be almost equal to those which favor Ravi.

However, when multiple conflict (a number occurs several times) occurs, our model needs several cycles to get rid of them but Ravi's model just throws them away. That probably is the reason why our steady state bandwidth is a little worse than his.

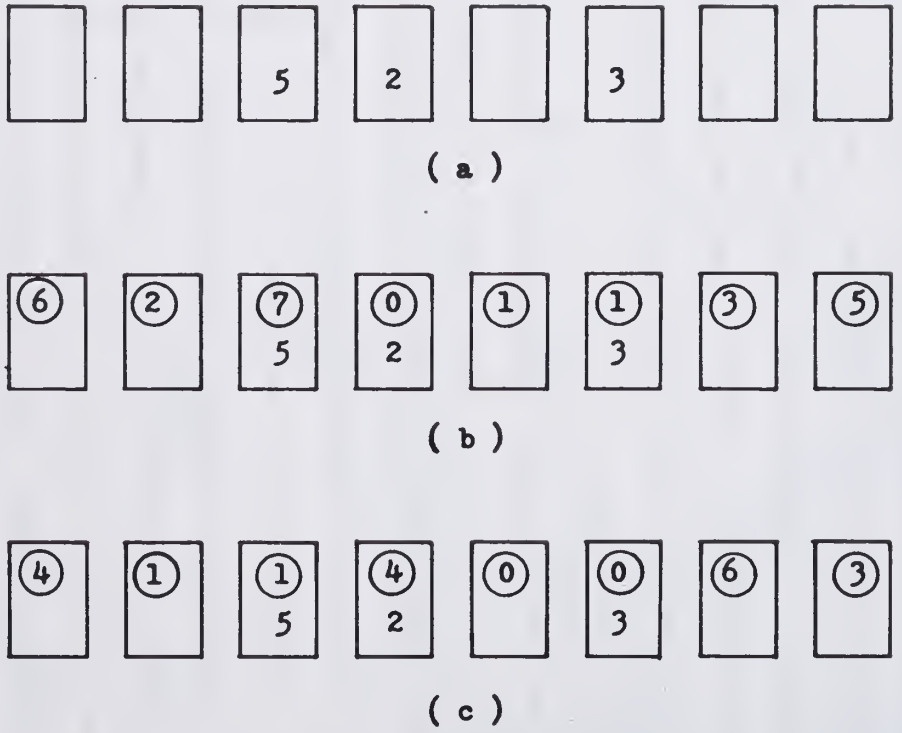


Figure 18. Comparison of Ravi's Model and Our Model III

Of course, we do not always increase both m and p at the same time. What happens if we hold one fixed and increase the other one? Figure 19 shows $B_{av.}^{ss.}$ versus m when p is fixed. The curve is very similar to Figure 4. That means, in real cases, bandwidth should grow as $p(1-e^{-cm})$ when the number and speed of p are fixed. On the other hand, when m is fixed the $B_{av.}^{ss.}$ against p the will have a similar curve.

The other parameter of this model is the queue length Q. However, both transient state bandwidth and steady state bandwidth show the fact that Q really does not make any difference. The reason is quite simple: if the request at the head of the queue is blocked, then the flow of requests is clogged. Generating a new request to pile up in the queue is the same as blocking the processor and generating the request later. So it is wasteful to build a long queue. One or two registers in the processor unit can do a good job already. This is why we did not put the parameter Q in all Model III diagrams. However, the use of longer queues allows us to do the address look-ahead. The contents of the queues can also be used as the decision factor of conflict resolution.

One more interesting thing is, our model I is essentially a special case of Model III with H = 1. So Model I is the lower bound of Model III. Thus the bandwidth of Model III is bounded above by Ravi's model and below by our model I.

As we said at the beginning of this chapter, this model is very useful to MIMD machines. The reason is that some processors might generate requests faster than the others. This asynchronism indeed ties our model to MIMD machines and some similar environments. Actually, if we add some extra control, Model III can also be used in SIMD machines, since Model I is a special case of Model III.

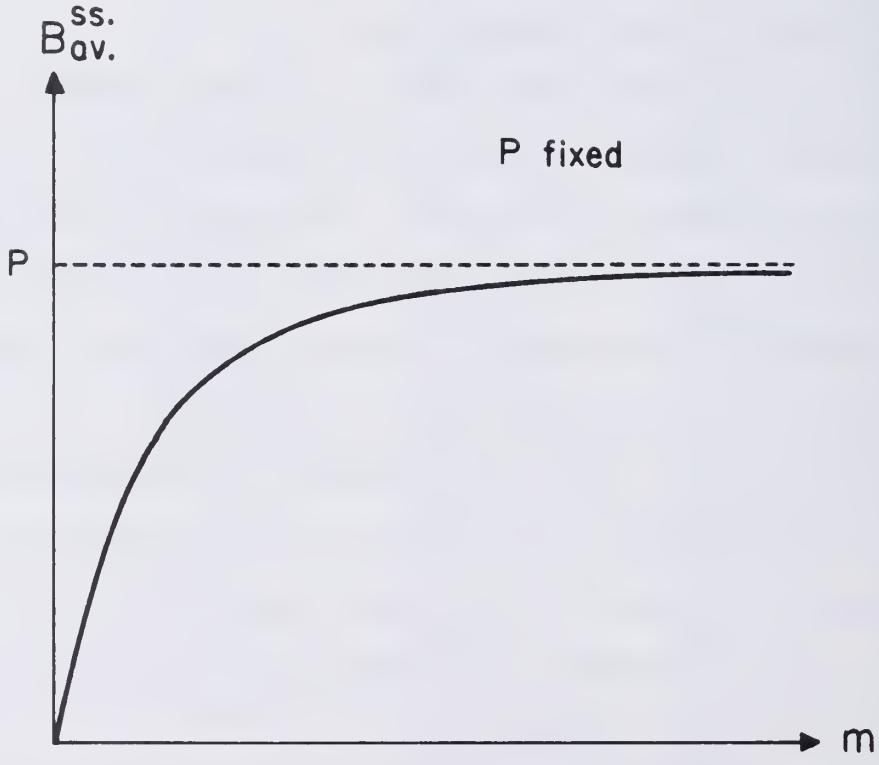


Figure 19. Steady State Bandwidth Curve when p is Fixed

4.5 Model IV: Queueing in the Memory

Our fourth interleaved memory model is very similar to the previous one except the queues are built in the memory modules instead of processor units. The logic structure is shown in Figure 20 which has p processors and m memory modules and each module contains a queue of length Q.

At the beginning of every cycle, each processor will generate a new request or regenerate the request which was blocked in the last cycle. Then all p requests will be sent down to a conflict resolution box. If more than one processor references the same memory module, the conflict resolution box will decide in what order these requests will go into the queue in their destination module. Then they will be gated into the queue one per clock pulse. So the clock rate should be carefully chosen in order to secure proper operation. If the queue does not have enough room to contain all these incoming requests, the conflict resolution circuit will block those unlucky ones at the end of the line.

When the memory module finishes the request of the last cycle, it will send out a completion signal which will push the queue down one place and the first request in the queue will enter the address register of the memory module. Then the memory module can begin another cycle and the whole thing starts again.

Again, we used a random number generator to simulate this model with all parameters having the same meanings as in Model III. Figure 21 shows the transient state bandwidth $B_{av.}^{T.}$ versus H for different m values and m = p. The dotted lines are for queue length 1 and solid lines for queue length 2. As you can see in this model, the longer queue will give

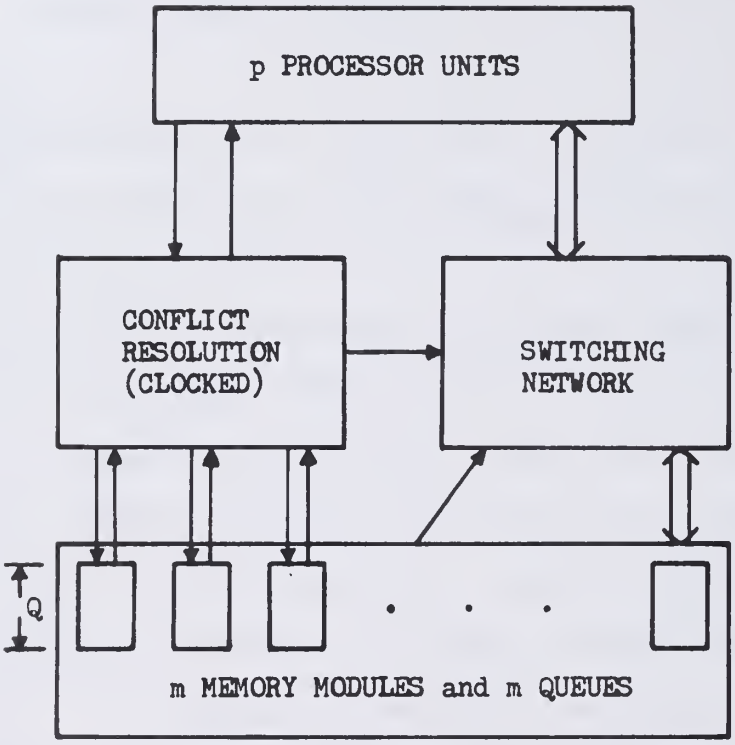


Figure 20. Block Diagram of Model IV

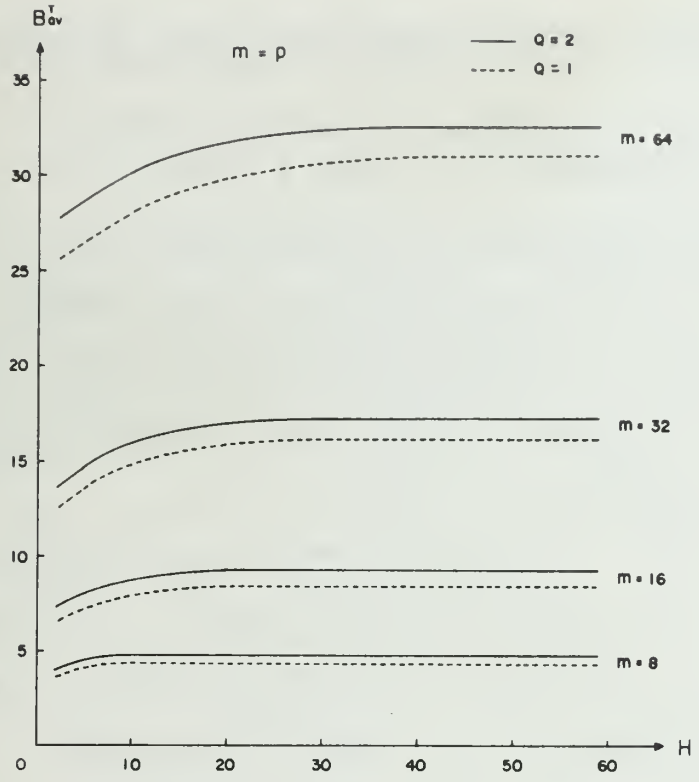


Figure 21. Transient State Bandwidth Curves of Model IV for Two Different Queue Lengths

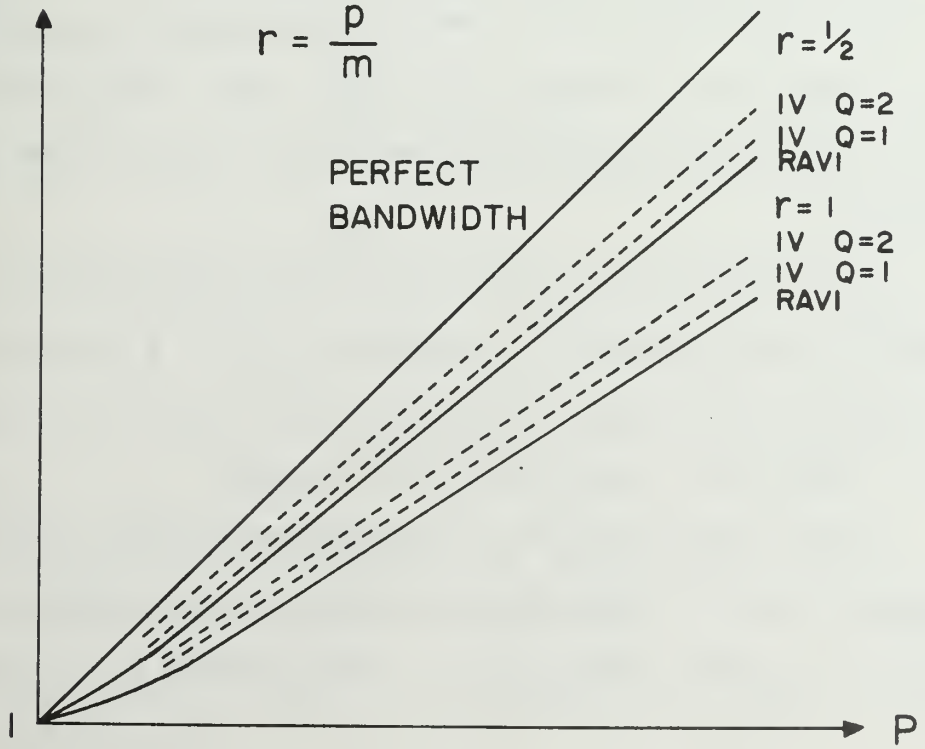


Figure 22. Steady State Bandwidth Curves of Model IV for Two Different Ratios of m and p

you better results. For other ratios of m and p, the curves have the same shape.

The histograms, i.e. B versus T curves, we have collected show that we can also improve our model IV by using a better conflict resolution algorithm. When using least-used-processor-first algorithm, we got almost 27% improvement over leftmost-processor-first algorithm. The improvement of the histogram is the same as we have shown in Figure 16. In Figure 21, we use the worst algorithm.

In Figure 22, we show the steady state bandwidth $B_{av.}^{ss.}$ for different ratios of m and p, again with Ravi's result. As you can see, even when queue length is 1, our model IV is better than Ravi's. Consequently, Model IV is better than Model III. The reason is very simple: no matter what the incoming requests are, the requests left in the queues can only enhance the bandwidth. So it is wise to build the queue in the memory module.

Figure 22 also reveals one thing: when we increase the queue length, the bandwidth curve moves upward and approaches the perfect bandwidth. One experiment shows that for infinite queue length the bandwidth curve moves very close to the perfect bandwidth. So the maximum possible bandwidth is p.

Just like Model III, this model is also lower bounded by Model I. Since Model I also corresponds to the special case H = 1 of this model. So the bandwidth might swing from the bandwidth of Model I to the perfect bandwidth. Both Q and H can increase the bandwidth.

However, this model has a very big drawback which could make this model useless, viz. the order that the requests will be satisfied is unpredictable. The time a request will be served depends on the place it

enters the queue. Hence a request might get satisfied earlier than its predecessors. If the data dependency is important, this model might not work properly.

One solution to this problem is to apply this model to a Tomasulo type machine. There all the data dependency relations will be handled in the processor by using the Tomasulo's algorithm. So there will be no data dependency between requests in the memory and the memory serving order is unimportant. Then our model IV will give a very good performance. However, this does substantially complicate the processor.

Besides, for read operations, this model has a big problem that no other one has, that is, more than one data fetched might go back to the same processor. This causes a two-way conflict problem which might seriously degrade the system performance.

So, this model unfortunately is not very useful. The performance we showed did not take these into consideration since we only wanted to show the potential of this model.

When Q = 1, this model still might be used in a MIMD machine. This is due to the fact that the instruction addresses and data addresses are interleaved in real programs. In the worst case, two consecutive requests will be served at the same, but this still preserves the proper order of execution. However, we then need a very complicated switching network to switch two pieces of information back at the same time in order not to degrade the performance. So the cost will be very high.

When we delete the queues from our model IV, or equivalently make Q = 0, this model essentially becomes Model III. Our simulation result also shows that when Q = 0 the steady state bandwidth is indeed the same as Model III.

5. COMPARISON OF ALL MODELS

5.1 Comparison of Performances

In order to compare the performances, we summarize in one graph the bandwidth curves of all the models we have discussed. Figure 23 shows the bandwidth versus p curves when m = p. Of course, the perfect bandwidth (slope = 1 line) is the best we can ever achieve, no model will pass this line. Also, nobody will fall below the bandwidth = 1 line which is the worst case for any model. All the performances lie between these lines.

Line (a) is the analytical result of Model I when m = p. Analytical results can be viewed as the steady state bandwidth. As we explained in the last chapter, this line is the lower bound for both Model III and Model IV, since it corresponds to the special case (or the worst case) of H = 1. Line (e) is the analytical result of Ravi's model when r = 1 ($B_{av} = (1- {}^{1}/e) *p$) which is the upper bound for Model III. So the performance of our model III swings from line (a) to line (e). If you recall, the queue length of this model does not influence the bandwidth. So when we increase H, the number of requests each processor can generate, the bandwidth curve will move from line (a) to line (b) and then to line (d) when H becomes very large. Line (b) is the transient state bandwidth when H is only 5 and line (d) is the steady state bandwidth of Model III. Line (f) is the steady state bandwidth of Model IV with queue length 1. This line is above Ravi's line. When we increase the queue length, the bandwidth curve keeps going up until reaching line (g) which corresponds to the infinite queue length. So the bandwidth of Model IV ranges from line (a) to line (g) depending on H and Q.

From this diagram, you can get a rough idea of how good these

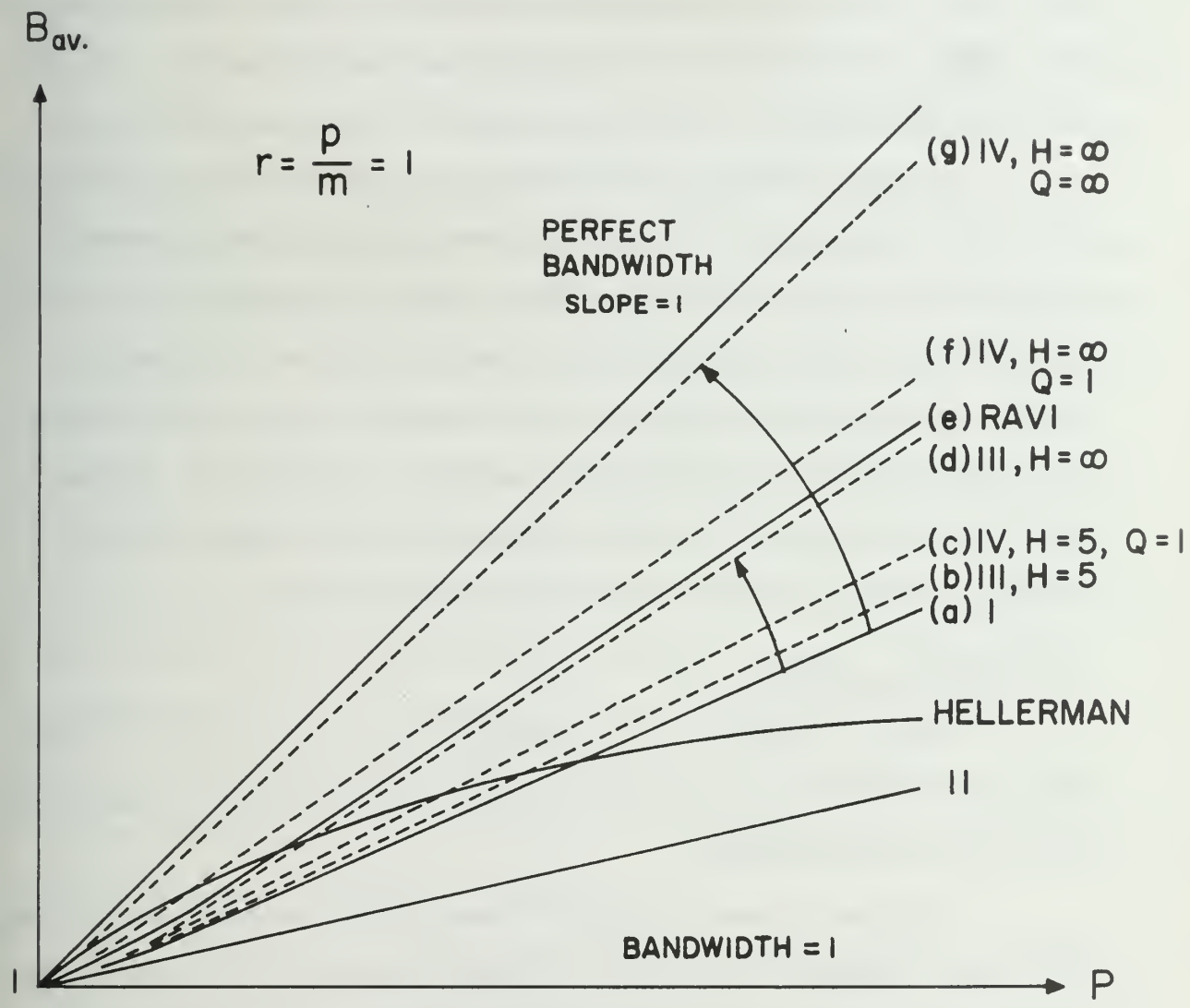


Figure 23. Bandwidth Curves of all Models

models can be and what happens if some of the parameters are changed. For other ratios of m and p, the relative positions of these curves remain the same. When r gets smaller (more memory modules than processors), all the curves move up, and when r gets larger, all the curves move down.

The most important thing about this diagram is that all the bandwidth curves, steady state or transient state, are linear with respect to p when we hold the ratio of m and p fixed. This is also true when we plot them against m. This contradicts the often quoted square root result accepted by people for a long time. This linearity has important implications for the design of multiprocessor machines. In Figure 23, we also plot Hellerman's curve. As you can see, all the other models obey the linearity principle instead of the square root principle.

5.2 Comparison of Costs

Another important thing we should consider is the cost to implement a model. When we are choosing a model, we must consider the trade-off between cost and performance.

Hellerman's model only needs a very simple control since it stops at the first conflict. So the cost of this model will be very cheap. But this way of handling conflicts is indeed the reason of bad performance.

Burnett, Coffman and Snowdon's model uses one more queue and a slightly more complicated control than Hellerman's model. So the cost will be a little bit higher. However, the performance has been greatly improved due to queueing the conflicts.

As we said in Chapter 3, these two models can only be used in a Tomasulo type machine where no data dependency problem is involved in handling the requests. This needs very complicated processors and hence

the total cost of the whole system will be increased accordingly. In Table II, we only consider the cost for the control portion.

Since Ravi did not consider how to handle the conflicts, his model becomes unrealistic. So we are not interested in implementing this model. However, there are a lot of similarities between this model and Model III, thus we may think our model III is a realistic realization of Ravi's model.

Although our model I displays the worst bandwidth and utilization among Models I, III and IV, the implementation of this model is the easiest and cheapest. Since there is no queue involved in this model, no extra hardware register and control circuit are needed. The only thing we need here is to build a relatively simple conflict resolution circuit. Since all p requests must be satisfied before new requests can be generated, there is no need to build a fancy conflict resolution circuit. The simplest circuit proposed in the next chapter can be used for this purpose.

As we said before, Model II has the worst performance and the implementation is not cheaper. Besides, all Model II can do Model I can do too. So Model II should not be considered.

Model III is a fairly good model in both performance and cost. Most processors have several hardware registers in them which can be used as an address queue after adding some control circuit. So this model seems to be the most plausible one. As we mentioned before, the bandwidth can be improved further by using fancier conflict resolution circuit.

From a bandwidth point of view, Model IV is the best model. But several reasons prevent us from choosing it. The first reason is that it is very expensive to build queues in the memory due to extra registers and control circuit. Second, it is very complicated and expensive to build

the conflict resolution box and the switching network. Third, the usefulness of this model is very limited when $Q \geq 2$. So we do not favor this model despite of its excellent performance.

Table II below shows the performances and the costs for all models. As you can see, Model III might be the best choice in the sense of cost effectiveness and usefulness.

Model	Data Dependency Type	Performance	Cost	Usefulness
Hellerman	D	Very bad	Very cheap	Tomasulo type machines
Burnett, Coffman and Snowdon	D	Good	Cheap	Tomasulo type machines
Ravi	C	Good	-----	-----
I	B	Bad	Cheap	SIMD machines
II	B	Very bad	Expensive	SIMD machines
III	C	Good	Cheap	1. MIMD machines 2. SIMD machines with extra control.
IV	D	Very good	Very expensive	1. MIMD machines when $Q = 1$. 2. SIMD machines when $Q > 1$ for some applications. 3. Tomasulo type machines

Table II. Comparisons of All Models

5.3 Data Dependency Types and Usefulness

In Chapter 3, we defined four data dependency types and classified other people's models accordingly. In the last chapter, we explained why our models I and II belong to type B, model III belongs to type C and model IV belongs to type D. We repeat the classification in the second column of Table II.

The data dependency is the major factor that decides the usefulness of a model. Just as we said before, no model is universally good for any kind of machine. If we want to use some model in full power, we must consider the data dependency type it fits best. So before we choose a model, we should decide what type of data dependency takes the major role in our machine. In Table II, we also list the machine type for each model. This essentially depends on the second column.

6. LOGIC DESIGN

6.1 The Problems

All our models shown in Chapter 4 have four basic components: processor units, conflict resolution box, switching network, and memory modules. We have shown their structures in Figures 6, 9, 12 and 20. The conflict resolution box and the switching network are the most important parts which together control the operation of the whole system.

Two interesting things about these logic structures are: how the conflict resolution box handles the conflicts, and how the switching network uses the resolution result as control information to switch the requests. Of course, the complexities depend on the model.

As we said before, there are a lot of tricks you can play in conflict resolution. We mentioned three strategies in Chapter 4 and we are going to show the designs of two of them in the next section. Any one of these conflict resolution circuits can be fitted into any model.

6.2 Circuit Design of Conflict Resolution Box

Since every processor might reference any memory module independently, we must use m identical pieces of circuit to meet all the possible cases. Each piece is associated with a memory module which will handle the conflicts happening in this module. In the worst case, all p requests will reference the same memory module, so every piece of conflict resolution circuit should have p inputs. Also there should be p outputs where each output will tell the corresponding processor whether it will be honored by this memory module or not. At any time, at most one output is active.

So the problem is to design a combinational circuit with p inputs and p outputs, such that when some of the inputs are active, only one of their corresponding outputs will be active. By active, we mean a certain kind of signal occurs at the input or output port, which might be a level signal or pulse signal.

Of course, the solution to this is not unique. For a different conflict resolution algorithm, you will get a different circuit design. We show two different designs below, one for leftmost-first algorithm and the other for random-selection algorithm.

6.2.1 Leftmost-first Circuit

Figure 24 shows a conflict resolution circuit which will signal a 1 at the output corresponding to the leftmost input that is active. At the beginning of a cycle, those requests who want to reference this module will send a signal to the corresponding input port. In Figure 24, we also show a possible way of doing this, i.e. attaching a decoder to each address register to decode the first few bits which represent the memory module number. Then a level signal will be sent to the proper conflict resolution circuit.

When this circuit starts working, a sense signal will be sent into the leftmost AND gate by the control unit. If I_j is the leftmost input port that is active, or has an input 1, then $S_1 = S_2 = \ldots = S_j = 1$ and $O_1 = O_2 = \ldots = O_{j-1} = 0$ due to $I_1 = I_2 = \ldots = I_{j-1} = 0$. Hence O_j will be 1 since both I_j and S_j are 1. But I_j will cause all of S_{j+1} to S_p be 0 which will cause all of O_{j+1} to O_p be 0.

Of course, this circuit works. But just like ripple carry adder, it needs p gate delays to propagate the signal through the circuit in the worst case. So it is not very attractive although it only takes 3p gates per memory module, or at total of O(mp) gates.

However, one can easily get the following set of equations for outputs:

$$O_1 = I_1$$

$$O_2 = \bar{I}_1 I_2$$

$$\vdots$$

$$O_p = \bar{I}_1 \bar{I}_2 \cdots \bar{I}_{p-1} I_p$$

So we can build an AND-gate tree to speed up this circuit. This will take O(p) gates per memory module but with only $\log_2 p$ gate delays. The implementation is easy and we omit it here.

6.2.2 Random-selection Circuit

As we mentioned in Chapter 4, the leftmost-first algorithm always displays the worst result. One algorithm that shows the best improvement is the least-used-first algorithm. But in order to find out who is the least used processor, we need a lot of sorting circuits or comparison circuits. This will cause tremendous cost and so we do not favor this scheme.

The other one that shows only a little worse performance than the least-used-first algorithm is the random-selection algorithm. The implementation of this scheme is very similar to the circuit in Figure 24. Figure 25 shows the circuit that uses random selection.

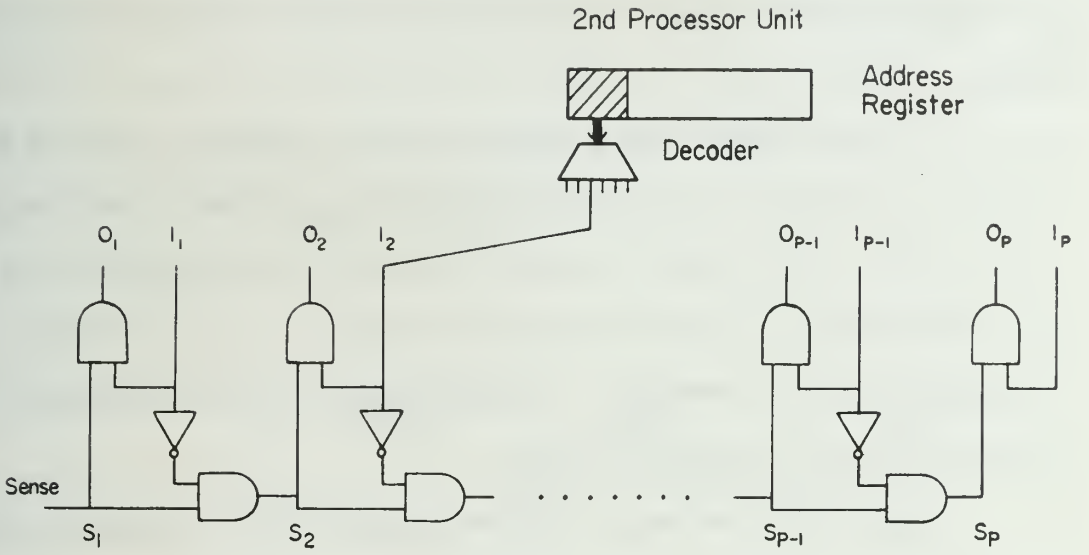


Figure 24. Conflict Resolution Circuit by Using Leftmost-first Algorithm

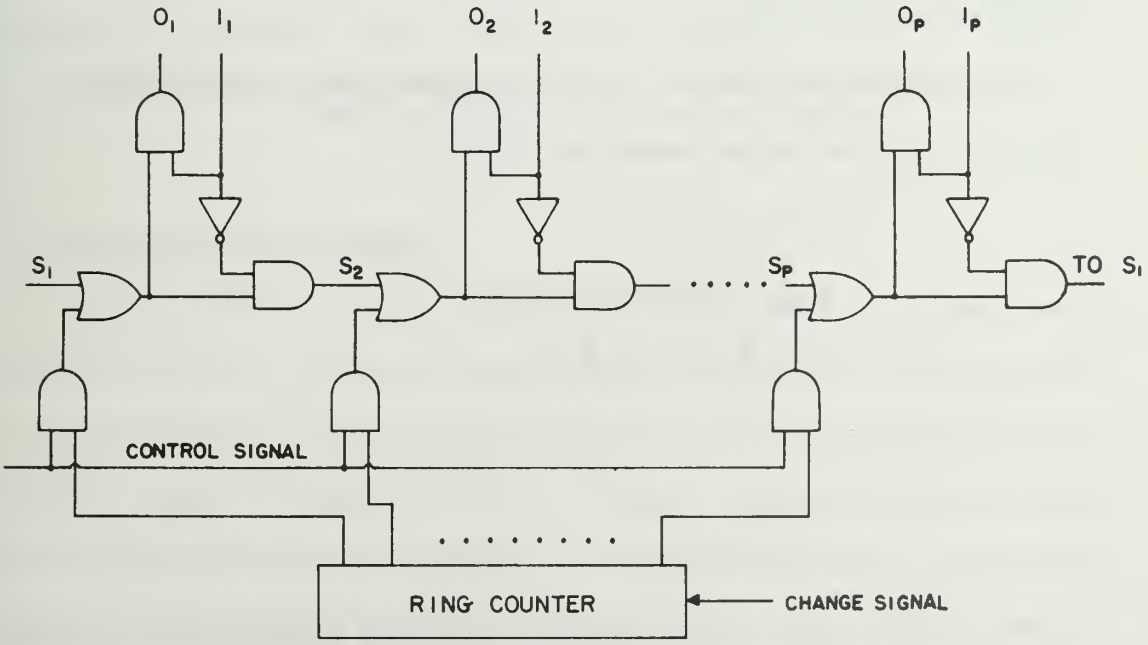


Figure 25. Conflict Resolution Circuit by Using Random-selection Algorithm

Here we just replace the sense input of Figure 24 by a p flip-flop ring counter. Only one output of this ring counter will be 1 at any time. The outputs of ring counter will be OR-ed with S_i's through control gates.

When control signal comes, only one output of those OR gates will be 1 according to the content of the ring counter. Then the closest input with a 1 will have a 1 at the output port, and the rest will all be 0. Equivalently, we just move the sense input to a point decided by the content of the ring counter. The content can be changed arbitrarily, so we have a random selection circuit.

The ring counter is just an end around shift register. If we shift the counter one place per cycle, we have something very similar to the polling scheme for handling interrupts.

This implementation would take 5p gates plus a ring counter per memory module. The gate delay is p. We can also use the tree connection to speed up this circuit to $\log_2 p$ gate delays. However, the output equations are more complicated than those for leftmost-first circuit. Here we only show the output equation for O_1:

$$
\begin{aligned}
O_1 = {} & I_1 S_1 \\
& + I_1 \bar{I}_2 \bar{I}_3 \cdots \bar{I}_p S_2 \\
& + I_1 \bar{I}_3 \bar{I}_4 \cdots \bar{I}_p S_3 \\
& \vdots \\
& + I_1 \bar{I}_p S_p
\end{aligned}
$$

Other output equations are similar except the subscripts are permuted symmetrically. Obviously, this equation will take O(p) gates and hence

the whole circuit will take $O(p^2)$ gates.

Figures 24 and 25 only show the conflict resolution circuit for one memory module. As we said earlier, a conflict resolution box takes m identical copies of this circuit, so the gate count will be m times that we gave above. Table III summarizes the gate delays and the total gate counts for the four possible designs.

Algorithm	Connection Type	Gate Delays	Total Gate Count
Leftmost-first	Ripple	$O(p)$	$O(mp)$
Leftmost-first	Tree	$O(\log_2 p)$	$O(mp)$
Random-selection	Ripple	$O(p)$	$O(mp)$ plus m ring counters
Random-selection	Tree	$O(\log_2 p)$	$O(mp)$ plus m ring counters

Table III. Gate Delays and Total Gate Counts for the Four Conflict Resolution Box Designs

6.3 Switching Network Design

The outputs of the conflict resolution box are not only used to signal the processors units whether their requests have been accepted or not, but also used to give the switching network control information for switching requests to memory modules. We have just described the detail of the conflict resolution circuit, so we will treat it as a well-known black box here in order to simplify the picture.

The switching problem in our system is not as simple as we might

think. Although we can use any non-blocking alignment network, such as a crossbar network, some control problems will make them inadequate for our systems.

In this section, we are going to show the design of a switching network by using another method, namely, the jam transfer method. This is a very simple and easy-controlled method, although it takes a lot of gates.

The typical configuration of jam transfer is shown in Figure 26. When control signal C is given, the content of flip-flop A will be gated into flip-flop B. In our system, this control signal is provided by the output of the conflict resolution circuit. The OR gates in front of the flip-flop B allow different resources to be connected to B.

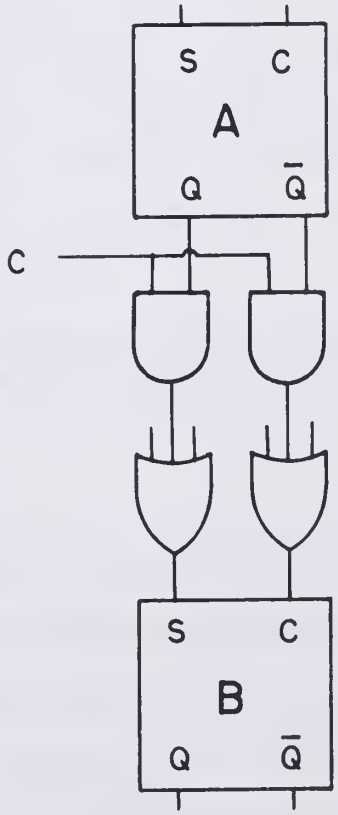


Figure 26. Jam Transfer

All the implementations of switching network for our models are very similar, we only show the system layout of Model I in Figure 27. The other models have the same basic connection but with fancier control circuit, so we only describe the difference in words.

In order to simplify the picture, we only show the connection between two processor units and one memory module. The rest can be connected in the same way. Again, single lines in the graph indicate individual data or control lines and double lines indicate sets of lines. DR's represent data registers which hold the data that transfer from or to the memory modules. PC's are the address registers that hold the addresses. The first part of PC contains the memory module number and it is connected to a decoder that will send a signal to the proper conflict resolution circuit. Only the second part will be sent down to the memory. In order to provide the returning processor unit number we attach a tag to each address. AB is a bit which will tell whether the request in PC has been accepted by the memory or not. All rectangles in Figure 27 actually represent an array of gates.

At the beginning of every cycle, each processor will generate an address in PC. The decoder then decodes the first part of the PC and sends a level signal to the proper conflict resolution circuit. If this processor wins the contest, the second part of the PC and TAG will be gated into the memory. Then the memory cycle begins. The BUSY bit will be set to disable any change on MAR and at the same time the AB bit will be set to block the address in next cycle. At the end of the memory cycle, the memory module will generate a completion signal which resets the BUSY bit to allow new address to be gated into MAR. This completion signal is also used to gate the data back to the processor if it is a read operation.

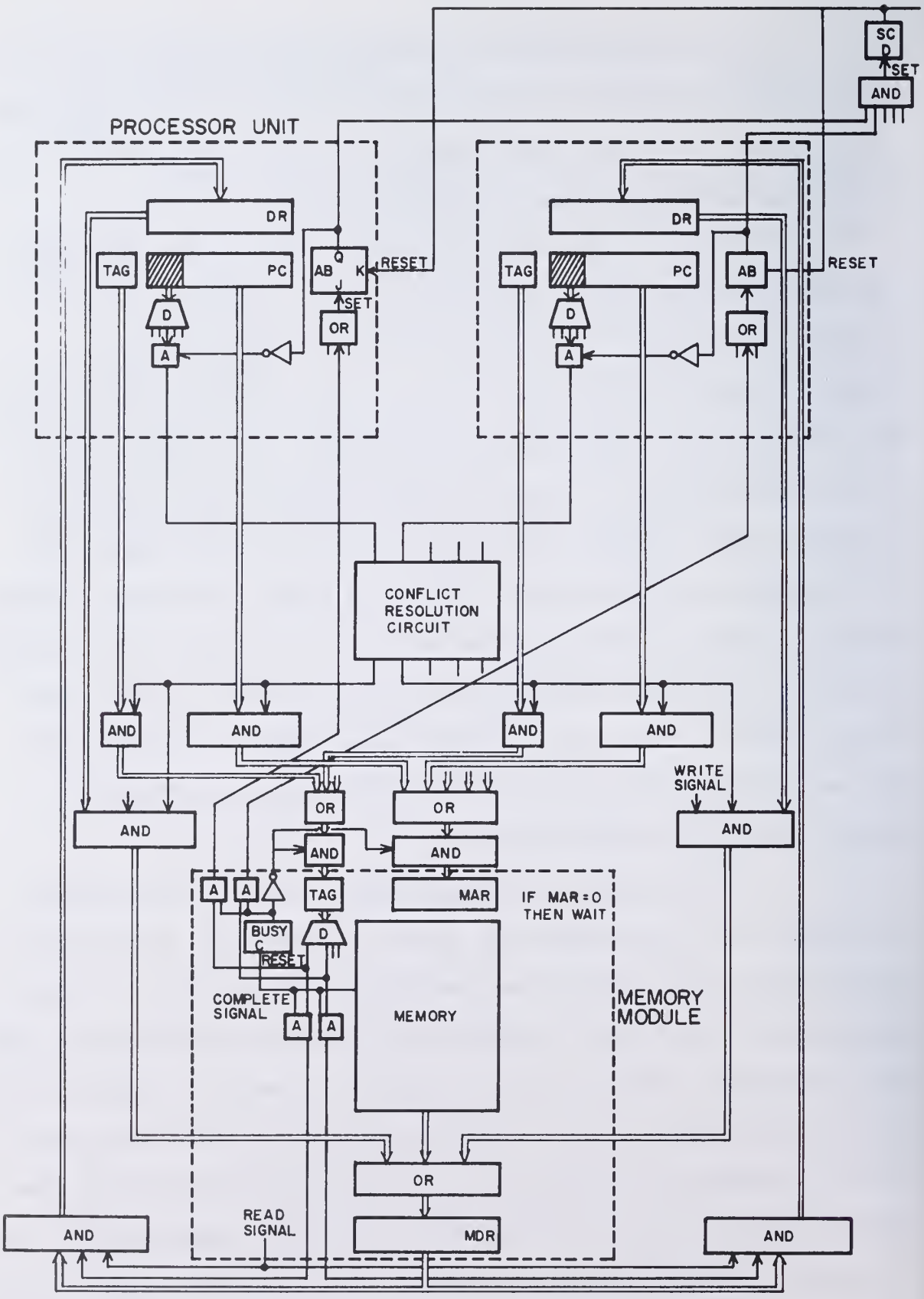


Figure 27. System Layout of Model I

After all AB's have been set, that means all the requests in PC's have been satisfied, the flip-flop SC will be set. This bit in turn resets all AB's and signals all the processors to generate new requests.

One thing we have not considered very accurately here is the synchronization of the control signals. This factor is extremely important in building a real machine.

For Model II, we can use the same method to design the switching network. However, the controls are different. If you recall, Model II uses a conflict detection box to detect all the conflicts. Then it uses the result to send all those requests that cause conflicts to a seperated area for later processing. This needs a more complicated control circuit. Since the usefulness of this model is very limited, and the design of control circuit is not the subject here, we will not show the layout and the design of this model.

The other two models differ from Model I in either the processor units or the memory modules. But the basic structures and the operations are very similar.

For Model III, we just replace every processor unit in Figure 27 by the module shown in Figure 28. Notice that the function of the AB bit is different now, since it is connected as a one-shot circuit which is used to shift the queue by only one place. Whenever the request in the PC is accepted by the memory, AB will be set which then generates a single pulse to shift down the content of the queue.

Differing from Model I, AB here will not prevent the processor from generating a new request when it is set. Instead we add another bit FULL to perform this job. FULL will be set by the control circuit of the

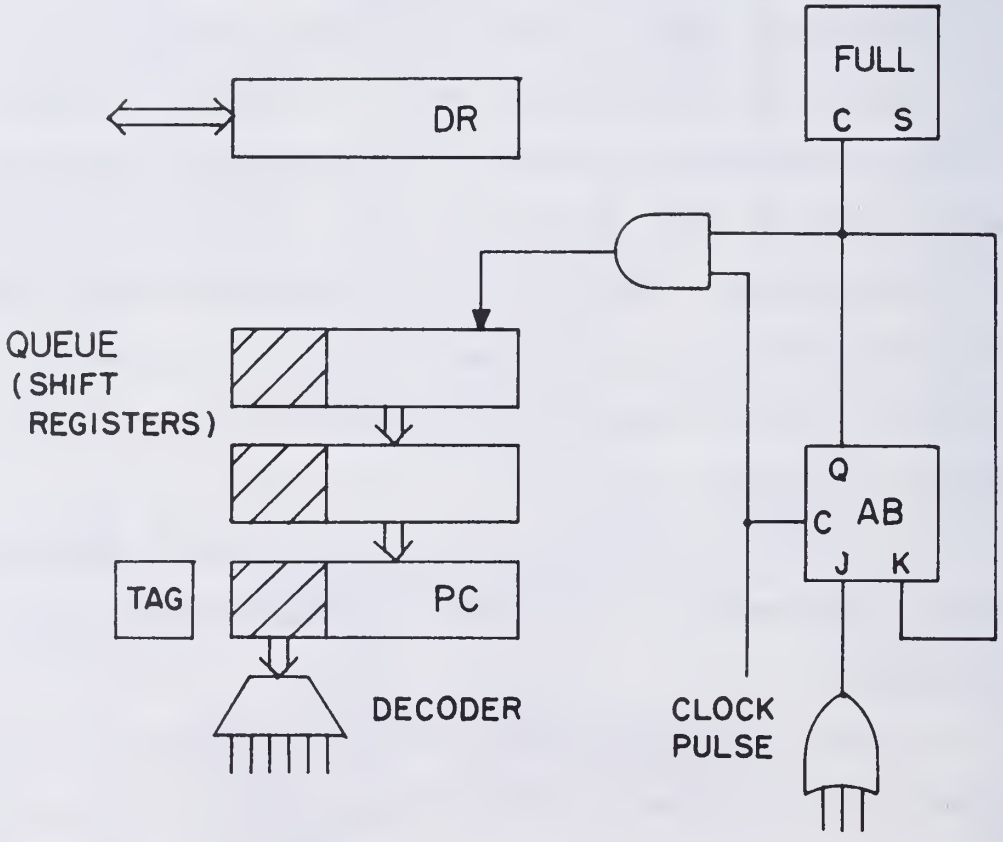


Figure 28. Processor Unit for Model III

queue and reset by the AB. In Figure 28, we ignore the control circuit for the queue.

The layout of Model IV is basically similar to that in Figure 27, except the queues are built in the memory. But as we said in Chapter 4, in order not to degrade the performance this model needs a two-way conflict resolution box and a complicated switching network. So the layout will be much more complicated. However, we still can use the same method as in Figure 27 to achieve the design of this model.

7. CONCLUSION

The early works by Hellerman and Burnett, Coffman and Snowdon were based on the fact that there is a large gap between processor speed and memory speed. For example, the CDC 7600 has a 27 ns processor cycle time and 270 ns memory cycle time. So they assumed that the processor is fast enough to supply enough requests to the system, and they worked on the single processor multiple memory modules system.

However, the modern memory technology has knocked this gap down to a factor of 5 or so. IBM system 370 models 135 and 145 use the bipolar technology which provides 150 ns memory cycle time. After packaging and power problems are solved, people will be able to use very fast bipolar memories. So the single processor system will be of less interest in the future and the multiprocessor multimemory system will become the main trend of computer design. That is why we are only interested in a model with multiple processors and multiple memory modules.

Our models actually assume that the processor speed is compatible with the memory speed since we only generate one request per processor per memory cycle. But we do take several things into consideration, for example the data dependency between requests, the control time we must spend for conflict resolution and switching requests, etc.. For a machine like TI's ASC which has 4 pipelines and 60 ns processor cycle time, 160 ns memory cycle time, this is not a bad assumption at all.

In real machines, the instructions and data are usually stored in such a way that the memory conflicts will be minimized. Several storage schemes have been proposed by Budnik and Kuck [15], Duncan Lawrie [16],

that allow you to fetch some data patterns of an array without conflict. So the probability of memory conflicts in real programs is actually reduced by using these schemes. If we apply our model in a real multiprocessor machine, the bandwidth we get will be higher. One reasonable way of simulating these models in order to get the "real" performances is to use some history tapes of actual programs as the inputs instead of random numbers. This has been done by Flynn [2].

No one else before has taken the data dependency problem into consideration when they designed their models. We agree that this is a very complicated problem. However, we believe that knowing the proper way of handling requests is the most important thing in designing memory systems. So we introduced four types of data dependency in Chapter 3. This is only a rough classification. More work needed to be done on this problem in order to get a better idea of how to handle requests.

The most important result of this thesis is the demonstration of a linear relationship between bandwidth and fixed ratios of the numbers of processors and memories in a multiprocessor machine, i.e. Figure 23. So when doubling the numbers of processors and memory modules, we should get twice as much bandwidth instead of $\sqrt{2}$ suggested by the traditional square root concept. This should be a revolutionary result in the memory system design.

APPENDIX

A. Expansion of Burnett and Coffman's Recursive Equation

Burnett and Coffman got a recursive solution for the generalized Hellerman model. The recursive function is repeated below:

$$C_m^0(0,k) = (m-1)_{k-1} - \sum_{j=1}^{k-1} \binom{k-1}{j} C_{m-j}^0(0,k-j) \qquad (A.1)$$

Actually, this equation can be expanded to be Stone's result (equation (2.7)). First, we introduce a combinatorial identity before we do the expansion, since we need it at every step.

$$\binom{n}{0}\binom{n}{i} - \binom{n}{1}\binom{n-1}{i-1} + \binom{n}{2}\binom{n-2}{i-2} - \ldots + (-1)^i\binom{n}{i}\binom{n-i}{0} = 0 \qquad (A.2)$$

Then start from equation (A.1). Expand the first term in the summation we get:

$$C_m^0(0,k) = (m-1)_{k-1} - \binom{k-1}{1} C_{m-1}^0(0,k-1) - \sum_{j=2}^{k-1} \binom{k-1}{j} C_{m-j}^0(0,k-j)$$

$$= (m-1)_{k-1} - \binom{k-1}{1}(m-1-1)_{k-1-1} - \sum_{j=2}^{k-1} \binom{k-1}{j} C_{m-j}^0(0,k-j)$$

$$+ \binom{k-1}{1} \sum_{j=1}^{k-2} \binom{k-2}{j} C_{m-1-j}^0(0,k-1-j)$$

The two first terms in the summations are $-\binom{k-1}{2}C_{m-2}^0(0,k-2)$ and $\binom{k-1}{1}\binom{k-2}{1}C_{m-2}^0(0,k-2)$. Combine them and apply equation (A.2) for n=k-1 and i=2, we get $\binom{k-1}{2}C_{m-2}^0(0,k-2)$. Repeat the substitution once we have:

$$C_m^0(0,k) = (m-1)_{k-1} - \binom{k-1}{1}(m-1-1)_{k-1-1} + \binom{k-2}{2}(m-2-1)_{k-2-1}$$

$$- \sum_{j=3}^{k-1} \binom{k-1}{j} C_{m-j}^0(0,k-j) + \binom{k-1}{1} \sum_{j=2}^{k-2} \binom{k-2}{j} C_{m-1-j}^0(0,k-1-j)$$

$$- \binom{k-1}{2} \sum_{j=1}^{k-3} \binom{k-3}{j} C^0_{m-2-j}(0,k-2-j)$$

All three summations have k-3 terms and their first terms are all of the same form $a*C^0_{m-3}(0,k-3)$. Their coefficients can be summed together to be $-\binom{k-1}{3}$ by using equation (A.2) for n=k-1 and i=3. Then substitute $-\binom{k-1}{3} C^0_{m-3}(0,k-3)$ by using equation (A.1) we generate $-\binom{k-1}{3}(m-3-1)_{k-3-1}$ and a summation.

If we keep going, we will produce $(-1)^j\binom{k-1}{j}(m-j-1)_{k-j-1}$ at the jth step. After k-1 steps, we get Stone's result:

$$C^0_m(0,m) = \sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j} (m-j-1)_{k-j-1}$$

B. The Proof of S_j

Although Stone gave the right solution to S_j, he did not really prove that his solution is right and how he got it. We must point out that the proof of S_j is not trivial. Here we show a way to derive this number. Again, S_j is the total number of sequences that are of length k and have at least j α-transitions.

First, let us release the restriction put on the first element of a sequence, that is, we allow it to be any number from 0 to m-1.

The total number of transitions in a k-length sequence is k-1. Of course, there are $\binom{k-1}{j}$ ways to select j positions for α-transitions. For a certain fixed selection, without loss of generality, we can assume that these j α-transitions are distributed into w disjoint groups and each group has u_i α-transitions. Hence $u_1+u_2+ \ldots +u_w = j$. By a group we mean those α-transitions that occur consecutively along the line. For example, let k=10 and j=6, then the selection $\beta\alpha\alpha\alpha\beta\alpha\beta\alpha\alpha$ has three groups (w = 3)

with $u_1=3$, $u_2=1$, and $u_3=2$.

Originally, we have m possible α-transitions to be chosen, namely, 01, 12, ... , (m-2)(m-1), (m-1)0. We can think of them as a circle of m numbers. Obviously, there are m ways to select a sequence of u_1+1 consecutive numbers (u_1 α-transitions) to be the first group. After that, we have $m-u_1-2$ α-transitions left to be used. Since no matter how we choose the first group, the two α-transitions at the ends can not be chosen by any other group. You can easily convince yourself of this by our definition of a group.

Now the circle has been broken and we can think of the remaining α-transitions as lying on a line. Our problem then is to choose w-1 groups from this line, no two adjacent to each other, and place them back into the sequence. Actually, the length of a group is not important since the first component of a group will decide the fate of the whole group. So we can think of our problem as finding out the number of ways of selecting w-1 objects, no two consecutive, from $m-u_1-2-\sum_{i=2}^{w}(u_i-1)$ or m-j+w-3 objects arrayed in a line, then permute them in w-1 positions. The reason we subtract $\sum_{i=2}^{w}(u_i-1)$ is to represent each group by its first element.

To find the number of such selections is actually a famous problem, namely, the Kaplansky Lemma, and the solution is:

$$\binom{(m-j+w-3)-(w-1)+1}{w-1} \quad \text{or} \quad \binom{m-j-1}{w-1}$$

Then we have (w-1)! ways to place them into w-1 positions. Since these w groups use w+j numbers, the number of ways to pad the remaining positions is (m-w-j)(m-w-j-1) ... (m-k+1).

Combine all these results, we have:

$$S_j = \binom{k-1}{j} m \binom{m-j-1}{w-1} (w-1)! (m-w-j) (m-w-j-1) \ldots (m-k+1)$$

$$= \binom{k-1}{j} m (m-j-1)_{k-j-1}$$

This is the same as Stone's S_j except with a factor m which comes from the fact that we assume the first element of a sequence can be any number from 0 to m-1. If the first element must be 0, S_j will be $1/m$ of the above result due to symmetry.

C. Simplification of Ravi's Bandwidth Equation

Ravi's bandwidth equation (equation (2.8)) is repeated here:

$$B_{av.} = \sum_{k=1}^{t} k \frac{k!\, S(p,k) \binom{m}{k}}{m^p} \qquad \text{where } t = \min(m,p) \qquad (C.1)$$

This equation can be simplified to a simple closed form of m and p only. Since the upper limit t has two different possible values, we will split the derivation into two different cases: one for t=m and one for t=p. Amazingly, the results of the two cases are the same.

First let t=p, or when $p \leq m$. Equation (C.1) becomes

$$B_{av.} = \sum_{k=1}^{p} \frac{k\, k!\, S(p,k) \binom{m}{k}}{m^p}$$

Let us take a look at the numerator. Since

$$k\, k! \binom{m}{k} = m (m-1) (m-2) \ldots (m-k+1) k$$

$$= m (m-1) (m-2) \ldots (m-k+1) (m-(m-k))$$

$$= m^2 (m-1) \ldots (m-k+1) - m (m-1) \ldots (m-k)$$

$$= m\, k! \binom{m}{k} - m\, k! \binom{m-1}{k}$$

so the numerator becomes:

$$m \sum_{k=1}^{p} \left(k! \binom{m}{k} - k! \binom{m-1}{k}\right) S(p,k)$$

$$= m \sum_{k=1}^{p} k! \binom{m}{k} S(p,k) - m \sum_{k=1}^{p} k! \binom{m-1}{k} S(p,k)$$

By the definition of Stirling number of the second kind:

$$m^p = \sum_{k=1}^{p} k! \binom{m}{k} S(p,k)$$

the above expression becomes

$$m^{p+1} - m(m-1)^p$$

Hence when $p \leq m$, equation (C.1) can be simplified to be:

$$B_{av.} = \frac{m^{p+1} - m(m-1)^p}{m^p}$$

$$= m\left[1 - (1-1/m)^p\right]$$

Now let $t=m$, or $p > m$. We can not play the same trick again since the upper limit is not p any more. So we need to find another way. One thing we can do is to substitute $k!S(p,k)$ by $\sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^p$, then we get

$$B_{av.} = \sum_{k=1}^{m} \frac{k\left(\sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^p\right) \binom{m}{k}}{m^p}$$

The upper limit of the summation in the numerator can be changed to k-1, since when $i = k$, $(k-i)^p$ becomes 0.

Now, if we just expand these two summations and place the $\frac{m(1+m)}{2}$ terms in an upper triangle (k=1, we get one term, k=2, we get two terms, etc..), then all terms on a diagonal have the same factor and the coefficients change regularly. So, if we start from the upper right corner and sum them diagonally, we will get expression (C.2) for the numerator.

$$m^{p+1} - m(m-1)^p + \sum_{i=2}^{m-1} a_i (m-i)^p \qquad (C.2)$$

where $a_i = \sum_{j=0}^{i} (-1)^j (m-i+j) \binom{m}{m-i+j} \binom{m-i+j}{j}$.

It is very easy to prove that a_i is equal to 0 for all i:

$$\sum_{j=0}^{i} (-1)^j (m-i+j) \binom{m}{m-i+j} \binom{m-i+j}{j}$$

$$= \sum_{j=0}^{i} (-1)^j (m-i+j) \binom{m}{i} \binom{i}{j}$$

$$= \binom{m}{i} \; m \sum_{j=0}^{i} (-1)^j \binom{i}{j} - i \sum_{j=0}^{i} (-1)^j \binom{i}{j} + \sum_{j=0}^{i} (-1)^j \, j \binom{i}{j}$$

$$= 0$$

since $\sum_{j=0}^{i} (-1)^j \binom{i}{j} = 0$ and $\sum_{j=0}^{i} (-1)^j j \binom{i}{j} = 0$.

So only the first two terms in expression (C.2) survive and the others become 0. Thus for $p > m$, $B_{av.}$ is the same as that for $p \leq m$, or

$$B_{av.} = m \left[1 - \left(1 - \frac{1}{m} \right)^p \right]$$

D. Derivation of $\lim_{p \to \infty} (1 - 1/m)^p = \frac{1}{e^r}$

Since $r = p/m$, then

$$\left(1 - \frac{1}{m}\right)^p = \left(1 - \frac{r}{p}\right)^p$$

$$= \frac{(p - r)^p}{p^p}$$

$$= \frac{(p-r)^{p-r} \, (p-r)^r}{p^p}$$

By using Stirling's formula $p! = p^p \, e^{-p} \sqrt{2\pi p}$, we can change the above expression to:

$$\frac{\sqrt{2\pi p}\ (p-r)!\ e^{p-r}\ (p-r)^r}{p!\ e^p\ \sqrt{2\pi(p-r)}}$$

$$= \frac{1}{e^r}\ \frac{p-r}{p}\ \frac{p-r}{p-1}\ \ldots\ \frac{p-r}{p-r+1}\sqrt{\frac{p}{p-r}}$$

as $p \to \infty$, all fractions except the first go to 1. So the limit of $(1-{}^1/_m)^p$ is ${}^1/_{e^r}$.

E. Solution of $f_n(m,s)$

$f_n(m,s)$ is the number of ways of putting n distinct objects into m distinct boxes with each box containing at most s objects. This is the coefficient of the $t^n/n!$ term of the following generating function:

$$\left(1 + t + t^2/2 + \ldots + t^s/s! \right)^m = \Sigma\, f_n(m,s)\ t^n/n!$$

We can derive the following recurrence:

$$f_n(m,s) = m\, f_{n-1}(m,s) - m \binom{n-1}{s} f_{n-1-s}(m-1,s)$$

For any set of n, m, s values, we can calculate $f_n(m,s)$ by using the above equation and the following facts:

$$f_n(m,s) = m^n \quad \text{if } n \leq s$$

$$f_{s+1}(m,s) = m^{s+1} - m$$

$$f_{s+2}(m,s) = m^{s+2} - m - (m)_2\ (s+2)$$

$$f_{s+3}(m,s) = m^{s+3} - m - (m)_2 \binom{s+4}{2} - (m)_3 \binom{s+3}{2}$$

F. Solution of $g_n(m,s)$

$g_n(m,s)$ can be said to be the complement function of $f_n(m,s)$. It

is the number of ways of putting n distinct objects into m distinct boxes with each box must contain at least s objects. The generating function is:

$$\left(e^t - 1 - t - t^2/2! - \ldots - t^{s-1}/(s-1)! \right)^m = \sum g_n(m,s)\, t^n/n!$$

The recurrence for $g_n(m,s)$ is:

$$g_n(m,s) = m\, g_{n-1}(m,s) + m \binom{n-1}{s-1} g_{n-s}(m-1,s)$$

For s = 2, the boundary conditions are:

$$g_n(m,2) = 0 \qquad \text{if } n < 2m$$

$$g_{2m}(m,2) = \frac{1}{2^m} (2m)!$$

$$g_{2m+1}(m,2) = \frac{1}{2^m} \frac{m}{3} (2m+1)!$$

G. IBM Random Number Generator

The random number generator we used in simulation is the IBM library routine RANDU. We show our version here:

```
INTEGER FUNCTION RANDU ( ISEED, M )
ISEED = ISEED * 65547
IF ( ISEED .LT. 0 ) ISEED = ISEED + 1073741824 + 1073741824
RANDU = ( ISEED * 0.4656613 E -9 ) * M
RETURN
```

The initial value of ISEED is 637823409 which is suggested by [14].

LIST OF REFERENCES

[1] Flores, I., "Derivation of a Waiting-Time Factor for a Multiple-Bank Memory," Journal of the ACM, Vol. 11, No. 3, pp. 265-282, July 1964.

[2] Sisson, S. S. and M. J. Flynn, "Addressing Patterns and Memory Handling Algorithms," AFIPS Conference Proceedings, 1968 Fall Joint Computer Conference, Vol. 33, Part 2, pp. 957-967, 1968.

[3] Hellerman, H., Digital Computer System Principles, pp. 228-229, New York, McGraw-Hill, 1967.

[4] Knuth, D. and C. Rao, "An Activity in the Interleaved Memory System," Unpublished Paper, 1975.

[5] Burnett, G. J. and E. G. Coffman, Jr., "A Combinatorial Problem Related to Interleaved Memory Systems," Journal of the ACM, Vol. 20, No. 1, pp. 39-45, January 1973.

[6] Stone, H. S., "A Note on a Combinatorial Problem of Burnett and Coffman," Communications of the ACM, Vol. 17, No. 3, pp. 165-166, March 1974.

[7] Liu, C. L., An Introduction to Combinatorial Analysis, pp. 110-111, New York, McGraw-Hill, 1968.

[8] Chang, D. Y., "Another Note on the Combinatorial Problem of Burnett and Coffman's Model," Unpublished Memo, 1975.

[9] Burnett, G. J. and E. G. Coffman, Jr., "A Study of Interleaved Memory Systems," AFIPS Conference Proceedings, 1970 Spring Joint Computer Conference, Vol. 36, pp. 467-474, 1970.

[10] Coffman, E. G., Jr., G. J. Burnett and R. A. Snowdon, "On the Performance of Interleaved Memories with Multiple-Word Bandwidth," IEEE Transactions on Computers, Vol. C-20, pp. 1570-1573, December 1971.

[11] Burnett, G. J. and E. G. Coffman, Jr., "Analysis of Interleaved Memory Systems Using Blockage Buffers," Communications of the ACM, Vol. 18, No. 2, pp. 91-95, February 1975.

[12] Ravi, C. V., "On the Bandwidth and Interference in Interleaved Memory Systems," IEEE Transactions on Computers, Vol. C-21, pp. 899-901, August 1972.

[13] Riordan, J., An Introduction to Combinatorial Analysis, New York, John Wiley and Sons, 1958.

[14] Richardson, B., "A Comparison of the IBM-SSP Random Number Generator with the Payne Feedback Shift Register Generator," CSO Document, University of Illinois at Urbana-Champaign.

[15] Budnik, P. and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Transactions on Computers, Vol. C-20, pp. 1566-1569, December 1971.

[16] Lawrie, D. H., "Memory-Processor Connection Networks," (Ph. D. thesis) University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 557, February 1973.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-747	2.	3. Recipient's Accession No.

4. Title and Subtitle

ANALYSIS AND DESIGN OF INTERLEAVED MEMORY SYSTEMS

5. Report Date: August 1975

6.

7. Author(s)

DONALD YI-CHUNG CHANG

8. Performing Organization Rept. No. UIUCDCS-R-75-747

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

US NSF DCR73-07980 A02

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered

Master's Thesis

14.

15. Supplementary Notes

16. Abstracts

High-speed computer systems usually organize their storage into several modules. Each module can operate simultaneously. Hence, several modules can be accessed at the same time. This effectively increases the throughput and the computation speed of the system.

In this report, we first describe several interleaved memory models designed and analyzed by other people. We repeat their results and also show some further improvements. We then present four new models and derive appropriate performance figures. The difference between previous models and our new models is the way we handle the requests and conflicts. Some of the circuit design problems are also discussed briefly in this report.

17. Key Words and Document Analysis. 17a. Descriptors

Conflict resolution
Data dependency
Interleaved memory system
Normalized bandwidth
Queueing
Steady-state bandwidth
Transient-state bandwidth

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Releast Unlimited	UNCLASSIFIED	88
	20. Security Class (This Page) UNCLASSIFIED	22. Price